D3.1 Secure Evolving Software Systems: a State of the Art Survey

Federica Paci, Fabio Massacci, Elisa Chiarani (UNITN) Michael Hafner (UIB), Zoltan Micskei (BME), Armstrong Nhlabatsi, Bashar Nuseibeh, Yijun Yu (OU)

Document information

Document Number	D3.1
Document Title	Secure Evolving Software Systems: a State of the Art Survey
Version	4.0
Status	Final
Work Package	WP 3
Deliverable Type	Report
Contractual Date of Delivery	31 July 2009
Actual Date of Delivery	31 July 2009
Responsible Unit	UNITN
Contributors	UNITN, BME, UIB, OU
Keyword List	Software Evolution, Security Requirements Engineering, Requirements evolution, Policies Evolution
Dissemination level	PU

Document change record

Version	Date	Status	Author (Unit)	Description
0.1	26 March 2009	Draft	B. Nuseibeh, Y. Yu (OU)	Outline
0.2	30 April 2009	Working	A. Nhlabatsi (OU)	SoA in Appendix B
0.3	17 June 2009	Working	A. Nhlabatsi (OU)	New version of SoA in Appendix B
0.4	23 June 2009	Working	M. Hafner (UIB) and Z. Micskei (BME)	Contribution to Appendix B and revision
1.0	24 June 2009	Draft	A. Nhlabatsi (OU)	SoA Report in Appendix B
2.0	16 July 2009	Draft	F. Paci (UNITN)	Draft of Introduction and Appendix C
2.1	27 July 2009	Working	A. Nhlabatsi (OU)	SoA Report new version in Appendix B
3.0	31 July 2009	Draft	F. Massacci (UNITN)	Revision of Introduction
4.0	31 July 2009	Final	E. Chiarani, F. Paci (UNITN)	Finalised version for deliverable

Executive summary

Long-lived software systems often undergo evolution over an extended period of time. Evolution of these systems is inevitable as they need to continue to satisfy changing business needs, new regulations and standards, and the introduction of novel technologies. Once the system is put in operation, new requirements emerge and existing requirements change. Parts of the software may have to be modified to correct errors that are found in operation, to adapt it for a new platform and to improve its performance or other non-functional properties.

Software systems inevitably have to change if they are to remain useful, but the change may undermine the security of the systems. It is thus important to design software systems that are evolvable and secure.

This report reviews the current approaches to software evolution, security requirements engineering, requirements evolution, evolution in access control, and presents new research strands in software evolution.

Index

1 Introduction

Software systems are undergoing continuing change and rapid revolution to respond to the changes in the environment, changes in user needs, developing concepts and advancing technologies [1]. Once software is put in operation, new requirements emerge and existing requirements change. Parts of the software may have to be modified to correct errors that are found in operation, to adapt it for a new platform and to improve its performance or other non-functional properties.

Software evolution, thus, is driven by changes that can affect the different artifacts produced during the software engineering process. The traditional WRSPM model [2] assumes five main artifacts:

- **domain knowledge (W)** that provides presumed environment facts;
- **requirements (R)** that indicate what the customer needs from the system, described in terms of its effect on the environment;
- **specifications (S)** that provide enough information for a developer to build a system that satisfies the requirements;
- a **program (P)** that implements the specification using the programming platform; and a **programming platform (M)** that provides the basis for programming a system that satisfies the requirements and specifications.

Changes in such model are based on the traditional waterfall model of software engineering [3] where a change in one of the artefact from an earlier stage are seen a cause of change in the artefact from later stages[1]:

Changes of the domain knowledge may result in changes to the requirements, the specification and the program associated with a system. Examples of domain knowledge changes include changes of the business processes associated with the system, upgrades to the hardware or operating system, changes in the platform, and changes in privacy and security laws and regulations.

Changes of requirements evolve because organizational, business and user needs change, because the operating environment change or because of errors and inconsistencies in the program. Changes of requirements often result in changes in system specification and program. For example, when requirements change also security policies change [4], and viceversa in order to preserve the security of the system.

Changes of the specifications are determined by changes in the requirements. They include changes to the software architecture, e.g the addition, the removal or the modification of system functionalities, the update or the modification of the security policies protecting the system.

Changes of the program are changes to the source code that may be the result of changes in the requirements and in the specification of the system.

[1] The changes to the programming platform can also be considered as changes of the domain knowledge.

Current research in software evolution has investigated evolution as change in specification and in the program, but evolution as change in requirements has received little attention from the research community. Another important aspect that has not been considered by the research community is how to support systems evolution while preserving the security of the systems. In fact, if on one side, software systems inevitably have to evolve to remain useful, on the other side, the change may undermine their security.

A further observation is that such model of waterfall change is also obsolete from the point of view of the classification of changes. We just make two examples here. At first we should consider not just the program and the programming platform as the “final” artefact but rather the full socio-technical system that is implemented [8]. A second issue is that changes might happen at different levels independently from each other (unless everything is considered a requirement). Changes in the program’s APIs might be due to changes in the threat model (countering new forms of buffer overflows) or due to upgrade of the operating systems. The reader will not find a new model in this report. This is the task of the conceptual model of SecureChange.

The report is organized as follows.

Appendix A is a background document, realized before the start of the project, that summarize the state-of-the-art about general requirements evolution, and discusses some of the research issues related to this topic. The main research strands identified are related to the identification of the causes dictating the change of requirements, the infrastructure for requirements evolution, and the design process. One of the major causes for requirements to change is the introduction of new laws and regulations for privacy, security, governance and safety. Thus, research efforts should be devoted to the developing of tools and techniques for systematically extracting requirements from laws and regulations in order to prove requirements compliance to such laws and regulations. An infrastructure for requirements evolution should include tools for version control, configuration management and visualization that have to accommodate the kinds of models used to represent requirements. Designs should be characterized by a high variability and modularity to be suited for evolution. High variability means supporting alternative designs for a given functionality. Highly modularity means that a system can have some of its components change with low impact on other components.

Appendix B is a focussed review on the notion of security requirements engineering. The appendix discusses open research issues and challenges that may need to be addressed in order to achieve the goal of security engineering for evolving software systems. One of the main challenges has been identified is the need for an approach for reasoning about both software evolution and security engineering. A cross fertilisation of approaches to managing software evolution with security requirements engineering is proposed as one way to address the problem of violating security requirements as result of evolution. Other challenges identified are designing change tolerant software systems, non-monotonicity of evolving software systems and secure evolution for adaptive software.

Finally, Appendix C analyzes which are the causes of evolution in access control systems and provides an overview of the approaches to manage their evolution.

References

[1] Lehman, M.M. and J.F. Ramil, *Software evolution: background, theory, practice.* Information Processing Letters, 2003. 88(1-2): p. 33-44.

[2] Gunter, C. A., Gunter, E. L., Jackson, M., and Zave, P. (2000). *A reference model for requirements and specifications*. IEEE Software, p 37-43.

[3] Sommerville I. *Software Engineering* 8th Edition. Pearson Education Limited.

[4] HE, Q., Anton, A. *Deriving Access Control Policies from Requirements Specifications and Database Designs*. Technical Report #TR-2004-24, September 2005.

[5] Zave, P. and M. Jackson, Four dark corners of requirements engineering. ACM Transactions on Software Engineering and Methodology, 1997. 6(1): p. 1-30.

[6] Nhlabatsi, A., B. Nuseibeh, and Y. Yu, Security Requirements Engineering for Evolving Software Systems: a Survey. 2009, The Open University: Milton Keynes. p. 27.

[7] Earst, N., Mylopolous, J., Wang, Y. Requirements Evolution and What (Research) to Do about It. Design Requirements Engineering: A Ten-Year Prospective. Springer, 2009.

APPENDIX A – Requirements Evolution and What (Research) to Do about It

Requirements Evolution and What (Research) to Do about It

Neil A. Ernst[1], John Mylopoulos[1,2], and Yiqiao Wang[1]

[1] University of Toronto, Dept. of Computer Science, Toronto, Canada
{nernst,jm,yw}@cs.toronto.edu
[2] University of Trento, Dept. of Information Engineering and Computer Science, Trento, Italy
jm@disi.unitn.it

Abstract. Requirements evolution is a research problem that has received little attention hitherto, but deserves much more. For systems to survive in a volatile world, where business needs, government regulations and computing platforms keep changing, software systems must evolve too in order to survive. We discuss the state-of-the-art for research on the topic, and predict some of the research problems that will need to be addressed in the next decade. We conclude with a concrete proposal for a run-time monitoring framework based on (requirements) goal models.

Keywords: Requirements, evolution, monitoring, satisfiability.

1 Introduction

It has been known for decades that changing requirements constitute one of the greatest risks for large software development projects [1]. That risk manifests itself routinely in statistics on failure and under-performance for such projects. "Changing requirements" usually refers to the phenomenon where stakeholders keep changing their minds on what they want out of a project, and where their priorities lie. Little attention has been paid to post-deployment requirements changes[1], occurring after a system is in operation, as a result of changing technologies, operational environments, and/or business needs. In this chapter we focus on this class of requirements changes and we refer to them as requirements evolution.

Evolution is a fact of life. Environments and the species that operate within them – living, artificial, or virtual – evolve. Evolution has been credited with the most advanced biological species that has lived on earth. The ability to evolve has also come to be treated as a prerequisite for the survival of a species. And, yet, evolution of the software systems species has only been studied at the level of code and design, but not at the level of requirements. In particular, there has been considerable research on software evolution, focusing on code reengineering and migration, architectural evolution, software refactoring, data migration and integration.

[1] . . . with the notable exception of research on traceability mechanisms. Of course, traceability is useful for evolving requirements, but doesn't actually solve the problem.

K. Lyytinen et al. (Eds.): Design Requirements Workshop, LNBIP 14, pp. 186–214, 2009.

However, the problem of post-deployment evolution of *requirements* (as opposed to architecture, design and/or code) hasn't made it yet into research agendas (see, for example, the topics that define the scope of a recently held workshop on "Dynamic Software Evolution", http://se.inf.ethz.ch/ moriol/DSE/About.html).

There are important reasons why requirements evolution is about to become a focal point for research activity in Software Engineering. The change from local, isolated communities to the global village isn't happening only for commerce, news and the environment. It is also happening for software systems. In the past, operational environments for software systems were stable, changes were local, and evolution had only local impact. Today, the operational environment of a growing number of software systems is global, open, partly unknown and always unpredictable. In this context, software systems have to evolve in order to cope ("survive" is the technical term for other species). Some of this evolution will be at the code level, and some at the architectural level. The most important evolution, however, will have to take place at the requirements level, to ensure that a system continues to meet the needs of its stakeholders and the constraints – economic, legal and otherwise – of its operational environment.

An obvious implication of the rise to prominence of requirements evolution is that the research to be conducted will have to be inter-disciplinary. Researchers from Management, Organizational Theory, Sociology and Law will have to be part of the community that studies root causes for change and how to derive from them new requirements. Evolution mechanisms and theories that account for them have been developed in Biology, Engineering, Organizational Theory and Artificial Intelligence. Some of these may serve as fruitful starting points for the research to be done.

A precondition for any comprehensive solution to the problem of evolving requirements is that design-time requirements are properly captured and maintained during a system's lifecycle, much like code. Accordingly, we (optimistically) predict that the days of lip service to requirements are coming to an end, as Software Engineering Research and Practice opt for lasting technical solutions in a volatile world. Growing interest in topics such as autonomic software, semantic web services, multi-agent and/or adaptive software, peer-to-peer computing (... and more!) give some evidence that this optimism is not totally unwarranted.

The main objective of this chapter is to review the past (section 2) and suggest a research agenda on requirements evolution for the future (section 3). After a general discussion of topics and issues, we focus on one item of this agenda – monitoring requirements – to make the discussion more concrete. The remainder of the paper presents some of our on-going work on the problem of monitoring requirements and generating diagnoses. Technical details of this work have been presented in [2].

2 The Past

In the area of software evolution, the work of M. Lehman [3] stands out, with a proposal backed by empirical data for laws of program evolution. These laws offer a coarse grain characterization of types of software and the nature of its

evolution over its lifetime. Lehman's work on program evolution actually started with a study of the development of OS/360, IBM's flagship operating system in the late 60s. The study found that the amount of debugging decreased over time and concluded that the system would have a troubled lifetime, which it did. A few years later, Fred Brooks (academic, but also former OS/360 project manager) excoriated the IBM approach to software management in his book "The Mythical Man Month" [4]. Using Lehman's observations as a foundation he formulated his own "Brooks' Law": adding manpower to a late software project makes it later; all software programs are ultimately doomed to succumb to their own internal inertia. Fernandez-Ramil et al. [5] offers a comprehensive collection of recent research on the topic of software evolution.

As noted in the introduction, the focus of much of the research on software evolution has been on the code. Few software systems come with explicit links to requirements models. Pragmatically, it is simpler to understand system evolution by examining code artifacts – files, classes, and possibly UML diagrams. For example, Gîrba and Ducasse [6] present a metamodel for understanding software evolution by analysing artifact history. Their discussion pays little attention to the problem domain, likely because there is no clear way of reconstructing it. Similarly, Xing and Stroulia [7] use class properties to recapitulate a series of UML class diagrams to detect class co-evolution. Again, this study pays no attention to the causes of these changes, some of which relate to requirements.

We begin this section with a discussion of work that first identified the issue of requirements evolution, summarizing various attempts to characterize the problem using frameworks and taxonomies. We conclude with a look at current approaches to managing evolving requirements, including module selection, management, and traceability.

2.1 Early Work

When discussing the drivers behind an evolving model, and ways of managing that evolution, there are many definitions and terminologies in use. Various researchers have attempted to categorize the phenomena of evolving systems, the majority of whom come from the software maintenance community. The importance of requirements models throughout the software lifecycle has long been recognized. Basili and Weiss [8] reported that the majority of changes to a system requirements document were trivial, requiring less than three hours to implement. However, a few errors required days or weeks to resolve. Similarly, Basili and Perricone [9] report that of errors detected in a system during implementation, 12% were due to poor requirements (and 36% due to poor specifications). Rather than present an overarching temporal list, we categorize pertinent research into categories and draw distinctions between them. The majority of these papers present viable approaches to understanding the concepts involved. Where there are differences, they are typically the result of different perspectives.

Harker et al. [10] classifies requirements into:

1. enduring – core to the business;
2. mutable – a product of external pressures;

3. emergent – surfaced during thorough elicitation;
4. consequential – identified after product implementation;
5. adaptive – requirements that support system agility; and finally,
6. migration requirements – those which help during the changeover.

Where Rajlich and Bennett [11] and Lientz and Swanson [12] (see below) are discussing the process (actions) of managing these changing requirements, Harker et al. are focusing on the structure of those requirements. There are many terms one might apply to change and evolution in software. Rowe et al. [13] define evolvability as "a system's ability to accept change", with the addition of the constraints that it be a least-cost change, as well as one preserving the integrity of the architecture. It isn't made clear why the preservation of architectural form is important – perhaps for backwards compatibility.

They mention four properties of evolvability: generality, adaptability, scalability, and extensibility. There is a two-way relationship among these. From generality to extensibility there is an increasing amount of change required for a given requirement; from extensibility to generality there is a increasing amount of up-front cost. In other words, to build an extensible system is initially cheap, but costly when the change needs to be made, since radical extensions to the architecture are required. This dimension characterizes a given system in terms of an architectural state space – similar to the 'space of action possibilities' described in Vicente [14, p. 123].

Another state space model is covered in Favre [15], which presents a '3D' model of evolution. The three dimensions are *model abstraction* (model, metamodel, etc.), *engineering/implementation*, and *representation*. Each dimension has an associated series of stages, and Favre uses the intersection of these dimensions to map a particular product in a software space. For example, engineering stages consist of requirements, architecture, design and implementation – the traditional phases of software development. If we talk about a system at the meta-level of requirements, with an implicit representation, an example might be a conceptual metamodel such as the UML metamodel.

Favre suggests the importance of combining these orthogonal dimensions is for understanding how the various dimensions co-evolve. For example, it is important to consider whether the specification is co-evolving with the implementation, whether the modeling language is keeping pace with the technology, etc. As Favre concludes, it is important to remember that 'languages, tools, and programs evolve in parallel'.

To understand the motivations behind making changes to a system, a seminal work in the field of software maintenance is Swanson [16] (see also Lientz and Swanson [12]). They categorize software evolution into adaptive (environmental changes), corrective and perfective (new internal requirements) maintenance. Later work has added the notion of preventive maintenance. The context in which this work was done differs greatly from today; however, this division can be a useful way of understanding the nature of the changes in the environment which provoke reaction in the system.

Rajlich and Bennett [11] propose a different model, reflecting their belief that the post-delivery lifecycle is more complex than the term 'maintenance' reflects. They divide the post-delivery phase into four stages: evolution (major updates), servicing (corrective maintenance), phaseout, and closedown. Such a model reflects activities undertaken by companies like Microsoft and its Windows family of products. A requirements model is involved at the evolution (and possibly the servicing) stage. This model may be at odds with more agile development techniques, although there is a lack of research into the implications of agile techniques for software maintenance (although see Svensson and Host [17] for a preliminary assessment).

The process of managing change is also the subject of Nelson et al. [18]. They talk about flexibility in the context of business processes. Successful organizations (and their systems) exhibit adaptability, or the willingness to 'engage the unfamiliar'. Flexibility is the ability of such a system to handle change pressures and adapt. This is characterized as either structural or procedural. There are several determinants of each. Structural flexibility relies on modularity, or design separation; change acceptance, the degree to which the technology has built-in abilities to adapt; and consistency, the ability to make changes painlessly. Procedural flexibility is determined by the rate of response, system expertise (up-to-date knowledge), and coordinated action. Together, these characteristics define what it means for a system to adapt to a given change event. High levels of the preceding characteristics imply a high affinity to accommodate change.

Many proposed requirements engineering frameworks ignore change acceptance, relying on users to understand the nature of the change, and manually incorporate it. Buckley et al. [19] offer a taxonomy that describes the HOW, WHEN, WHERE and WHAT questions of software evolution (but not WHY or WHO). They suggest that such a taxonomy will help in understanding the mechanisms of the change, with a view to designing strategies for accommodating these processes. They categorize these questions into four dimensions of software change: change support, temporal change properties, object of change, and system properties. They analyze three tools which have seen evolution along the lines of the taxonomy. Requirements change is not specifically mentioned, but can be thought of as driving temporal change properties – e.g., a change in the environment will drive a change in the software.

In an attempt to bring together various software maintenance taxonomies, Chapin et al. [20] propose a high-level taxonomy for understanding the types of activities that occur in this area. The ontology is based on an impact model, examining evolution in the context of change to business processes and change to software (presumably this can be extended to refer to software-based system). They classify change events into a cascading, 4-part hierarchy of 12 categories, reflecting what they say is the wide diversity of concepts that exist in research and practice. Extending from perfective, adaptive, and corrective, they include four categories: support interface, documentation, software properties, and business rules.

For example, within business rules they define three changes: reductive, corrective, and enhancive. Their business rules category has the closest relationship to the concept of requirements. Changes in this category also have the highest impact on both software and business processes. According to this definition then, requirements changes will have the greatest cost for an organization. This idea certainly fits with the research findings suggesting that fixing requirements problems consistitute by far the largest cost in system maintenance (e.g., see Standish reports, although these are of uncertain research value). However, Chapin et al. do not explicitly discuss requirements. For example, they mention 'change requests', user-driven needs, as drivers, but make no reference to updated requirements. They also distinguish between maintenance – changes in the first 3 categories – and evolution, which (in their definition) primarily affects business rules. This is certainly the sense this chapter refers to.

Many of the prior papers mention requirements only because an implicit change in requirements has driven some corresponding change in the implemented software system. However, our research is concerned with the nature of these requirements changes. This was also the subject of research by Massimo Felici. In [21], he refers to requirements evolving in the early phases of a system, with perfective maintenance occurring toward the end of a system's lifespan. However, this view is at odds with the current view of requirements as something that exists throughout the project lifecycle.

In [22], the analysis begins with the observation that requirements frameworks generally do a poor job handling evolving requirements. The PROTEUS classication of requirements evolution (that of Harker et al.) is presented as a way to understand how requirements evolve. A requirement is either stable or changing. If the latter, it can be one of five subtypes: mutable, due to environmental factors; emergent, due to stakeholder engagement; consequential, resulting from the interaction of system and environment; adaptive, due to task variation; and migration, arising from planned business changes. This taxonomy of causes of requirements evolution is fairly concise yet comprehensive. Felici also discusses the similar causal taxonomy of Sommerville and Sawyer [23], which they term 'volatile requirements'. Sommerville and Sawyer use the categories of mutable, emergent, consequential, and compatibility requirements. Similarly, [24] presents the EVE framework for characterizing change, but without providing specifics on the problem beyond a metamodel.

2.2 Requirements Management

Requirements management studies how best to control the impacts of change on requirements. Properly managing change events — such as new stakeholder requirements — can be essential to reducing the amount of model evolution that occurs. A key research contribution in this area is a better understanding of how exactly these external pressures manifest themselves.

For example, Stark et al. [25] discuss change to requirements during the system release process. A release can be a minor version of an existing product, so this is a legitimate use of the term requirements evolution. They were responsible

for the development of missile warning software. The study produced some invaluable information on how change was occurring in the project: for example, 108 requirements were changed, and of this figure, 59% were additions (scope creep). They attempt to produce a predictive model of changes, but it isn't clear how generalizable such a model would be.

Similar research is reported by Basili and Weiss [8], in the context of another military project, the A-7 control software. They describe the nature of requirements changes on the project. The biggest issue seemed to be that many of the facts used in the requirements document were simply incorrect (51% of errors). They also categorize the errors from trivial to formidable. Although only one of the latter was encountered, it required 4 person-weeks of effort to resolve.

Lormans et al. [26] motivates a more structured approach to requirements management. They used a formal requirements management system, but encountered difficulty in exchanging requirement models with clients. Such 'models' were often in text form, or semi-structured representations. They propose a more elaborate management model that can address some of these challenges.

Wiegers [27] discusses four common tools for requirements management. To some degree each support the notion of managing evolving requirements. There is a question as to how well these tools reflect the reality in the code. Typically the tools store requirements as objects or relations, and then allow various operations, such as mapping to test suites or design documents. The biggest challenge is often maintaining traceability links between requirements and implementation. Roshandel et al. [28] discuss one approach for managing architectural evolution in sync with code. Another approach is to ignore everything but the source code, and reverse engineering requirements from there, as described in Yu et al. [29]. Finally, managing requirements will require configuration management tools similar to CVS, Subversion, and other code repositories. Tools like diff or patch need analogues in the model domain. Work in model merging, e.g., Niu et al. [30] will be important here.

Another emerging issue is the design of dynamic, adaptive software-based system. We discuss one approach to design such a system in section 4. Such systems are composed of multiple components, which may not be under one's direct control. Such systems are often categorized as Software as Service (SaaS) or Service-Oriented Architecture (SOA) domains. For these domains, we view requirements as the business drivers that specify which components, and in what priority, should be composed. A paper by Berry et al. [31] provides a useful 'four-level' characterization of the nature of the compositions and adaptations involved: the levels correspond to who (or what) is doing the requirements analysis: 1) the designer, on the domain; 2) the adaptive system, upon encountering some new condition; 3) the designer of the system, attempting to anticipate the nature of the second adaptation; or 4) a designer of new adaptation mechanisms.

Composing these components (or agents, or services) is an emerging research problem, and one in which requirements evolution will have a major role. Work on software customization Liaskos [32], for example, provides some insight into techniques for managing such composition, although it ignores the problem of

changes in the underlying requirements themselves. Related work in Jureta et al. [33] makes more explicit the idea that requirements cannot be fully specified prior to system implementation. They characterize this approach as one in which there is only one main requirement for the system, namely, that the system be able to handle any stakeholder requirement. Determining which stakeholder requirements are reasonable (i.e., within system scope) will be an important research problem.

Recent work has focused on Commercial Off-The-Shelf (aka COTS) components. A change in one component, driven by an evolution in a particular requirement, might impact other components. Etien and Salinesi [34] term this *co-evolution*. It is a challenge to integrate these COTS-based systems in such an environment:

> [COTS-based systems] are uncontrollably evolving, averaging up to 10 months between new releases, and are generally unsupported by their vendors after three subsequent releases. (Boehm [35, p. 9])

The work that led to that analysis, Yang et al. [36], discusses the issue of COTS-based software and requirements. They claim that defining requirements before evaluating various COTS options prematurely commits the development to a product that may turn out to be unsuitable. They argue for a concurrent development methodology that assesses COTS feasibility at the same time as developing the system itself. In other words, they argue for a spiral model approach (Boehm, 1988) to developing the requirements for such systems (not surprisingly). Nuseibeh [37] makes a similar point with his 'Twin Peaks' model. A requirements management tool that provided support for understanding the features, capabilities, and likelihood of change in various COTS products would be invaluable in such systems. Understanding how the requirements themselves might evolve would be one important aspect.

Traceability is an aspect of requirements management that identifies interdependencies between elements in the environment to elements within a system. Traceability is a necessary, but not a sufficient mechanism for managing evolving requirements. Without a link, the downstream impact of requirements changes will not be clear. Traceability can be divided into two aspects, after Gotel and Finkelstein [38]. One needs a trace from the various phenomena in the environment, to the specification of the requirements for the system. Once specified, a link should also be established between the specification and the implementation. The former case is relatively less studied, and is less amenable to formalization.

Requirements monitoring, first proposed in [39], and extended in [40], is one mechanism for tracing between requirements and code. Monitoring involves inserting code into a system to determine how well requirements are being met. A monitor records the usage patterns of the system, such as numbers of licenses in use. This information can be extracted and used to evolve the system, possibly dynamically. In this sense, monitors are quite similar to control instrumentation in, for example, industrial plants. This approach is promising, but does assume that requirements and environmental conditions can be specified accurately enough that monitoring is possible.

Traceability is more difficult with non-functional requirements, because by definition these requirements do not have quantitative satisfaction criteria. Cleland-Huang et al. [41] discuss a probabilistic information retrieval mechanism for recovering non-functional requirements from class diagrams. Three broad categories of artifacts are defined. A softgoal model is used to assess change impacts on UML artifacts, and an information retrieval approach is used to generate the traceability links between the two models. Ramesh and Jarke [42] give a lengthy overview of empirical studies of requirements traceability.

Monitoring also has a vital role to play in the design of autonomic systems ([43]). These are systems that can can self-repair, self-configure, self-optimize and self-protect. Of course, the ability to self-anything presupposes that such systems monitor the environment and their performance within that environment, diagnose failures or underperformance, and compensate by changing their behaviour.

3 A Research Agenda for 2020

So, assume that we have our operating software system and changes occur that need to be accommodated, somehow. The changes may be in the requirements of the system. For example, new functions need to be supported, or system performance needs to be enhanced. Increasingly, changes to requirements are caused by laws and regulations intended to safeguard the public's interests in areas of safety, security, privacy and governance. Changes may also be dictated by changing domain assumptions, such as increased workload caused by increased business activity. Last, but not least, changes may be dictated by new or evolving technologies that require migration to new platforms. New or evolving technologies can also open new opportunities for fulfilling business objectives, for example by offering new forms of business transactions, as with e-commerce and e-business.

Whatever the cause for a change, there are two basic approaches for dealing with it. The first, more pedestrian, approach to change has software engineers deal with it. This approach has traditionally been called software maintenance and it is generally recognized as the most expensive phase in a software system's lifecycle. A second approach for dealing with a change is to make the system adaptive in the first place, so that it can accommodate changes by using internal mechanisms, without human intervention or at least with intervention from end users only. The obvious advantage of this approach is that it makes change more immediate and less costly. Its main drawback, on the other hand, is that change needs to be thought out at design time, thereby increasing the complexity of the design. The recent focus on autonomic and/or adaptive software in the research community suggests that we are heading for automated approaches to software evolution, much like other engineering disciplines did decades ago.

Next, we list a number of research strands and discuss some of the problems that lie within their scope.

Infrastructure for requirements evolution. Research and practice on code evolution has produced a wealth of research concepts and tools. Version

control and configuration management, reverse engineering and visualization tools, refactoring and migration tools, among many. As indicated earlier, software of the future will consist not only of code and documentation, but also requirements and other types of models representing design, functionality and variability. Moreover, their interdependencies, for example, traceability links, will have to be maintained consistent and up-to-date for these artifacts to remain useful throughout a system's lifetime. Accordingly, the infrastructure for code evolution will have to be extended to accommodate these other kinds of artifacts. This is consistent with Model-Driven Software Engineering, as advocated by the Object Management Group (OMG).

Focusing on requirements, an infrastructure for requirements evolution will have to include tools for version control, configuration management and visualization. These tools will have to accommodate the kinds of models used to represent requirements. These models range from UML use cases that represent functional aspects of the system-to-be, all the way to goal models that capture stakeholder needs and rationalize any proposed functionality for the system-to-be. The problem of evolving traceability links from requirements to code has already been dealt with in the work of Jane Cleland-Huang and her colleagues (e.g., [44, 41, 45]).

Understanding root causes for change. We are interested here in characterizing generic root causes for change that dictate requirements evolution. For example, businesses are moving into network-based business models, such as service value networks and ecosystems. Such trends are bound to generate a host of new requirements on operational systems that will have to be addressed by requirements engineers and software reengineers. As another example, Governments around the world have been introducing legislation to address growing concerns for security, privacy, governance and safety. This makes regulatory compliance another major cause for requirements change. The introduction of a single Act in the US (Sarbanes-Oxley Act) in 2002 resulted in a monumental amount of change for business processes as well as software in business organizations. The costs of this change have been estimated at US$5.8B for one year alone (2005).

We would like to develop tools and techniques for systematically extracting requirements from laws and regulations. In tackling this research task, it is important to note that the concepts of law, such as "right" and "obligation", are not requirements. Consider a law about privacy that makes it an obligation for employers to protect and restrict the use of employee personal information stored in their databases. This obligation may be translated in many different ways into responsibilities of relevant actors so that the obligation is met. Each of these assignments of responsibility corresponds to a different set of requirements – i.e., stakeholder needs – that will have to be addressed by the software systems and the business processes of an organization.

This is a broad, inter-disciplinary and long-term research strand. Some research within its scope has already been done by Annie Anton, Travis Breaux

and colleagues, e.g., [46]. This is also the topic of Alberto Siena's PhD thesis, see [47] for early results.

Evolution mechanisms. Once we have identified what are the changes to requirements, we need to implement them by changing the system-at-hand. This may be done manually, possibly with tool support, by developing novel reengineering techniques. More interestingly, evolution may be done automatically by using mechanisms, inspired by different disciplines (Biology, Control Theory, Economics, Machine Learning, ...). Doing research along this strand will require much experimentation to evaluate the effectiveness of different evolution techniques.

A number of research projects are working on design principles for autonomic and adaptive software systems (see, for example, on-going series of ICSE workshops on *Software Engineering for Adaptive and Self-Managing Systems*, http://www.hpi.uni-potsdam.de/giese/events/2008/seams2008/). Many of these projects employ a monitor-diagnose-compensate feedback loop in order to support adaptation of a system in response to undesirable changes of monitored data. The inclusion of such a feedback loop in support of adaptivity introduces the problem of designing monitoring, diagnosis and compensation mechanisms in the architecture of software systems. Control Theory offers a rich set of concepts of research results on how to design such loops in the realm of real-time continuous processes. Unfortunately, the development of such a theory for discrete systems is still in its early stages (though work has been done, see for example [48]).

Design for evolution. Some designs are better suited for evolution than others. For example, a design that can deliver a given functionality in many different ways is better than one that delivers it in a single way. Such designs are said to have *high variability*.

Variability is an important topic in many scientific disciplines that study variations among the members of a species, or a class of phenomena. In fact, the theory of evolution as presented by Darwin [49] holds that variability exists in the inheritable traits possessed by individual organisms of a species. This variability may result in differences in the ability of each organism to reproduce and survive within its environment. And this is the basis for the evolution of species. Note that a species in Biology corresponds to a high variability software system in Software Engineering, while an individual organism corresponds to a particular configuration of a high variability software system.

Variability has been studied in the context of product families [50], where variation points define choices that exist within the family for a particular feature of the family. The space of alternative members of a family can be characterized by a feature model [51]. Feature models capture variability in the *design space* of a product family, or a software system for that matter. They tell us what configurations of features are consistent and can co-exist within one configuration. For example, variation points may arise from the operating platform on which a family member will run (Windows, Linux, MacOS), or the weight of the

functionality offered (personal, business, pro). *Problem variability*, on the other hand, focuses on variability in the problem to be solved. For instance, scheduling a meeting may be accomplished by having the initiator contact potential participants to set a time and location. Alternatively, the initiator may submit her request to a meeting scheduler who does everything. The alternatives here characterize the structure of the problem to be solved and have nothing to do with features that the system-to-be will eventually have.

Designing for variability through analysis of both the problem and design space will remain a fruitful area of research with Requirements Engineering. See [32] for a PhD thesis that focuses on problem variability.

Variability of biological species changes over time, as variants are created through mutation or other mechanisms, while others perish. We need comparable mechanisms for software through which the set of possible instances for a software system changes over time. In particular, it is important to study two forms of variability change: means-based variability, and ends-based variability.

Means-based variability change leaves the ends/purpose of a software system unchanged, but changes the means through which the ends can be achieved. For example, consider a meeting scheduling system that offers a range of alternatives for meeting scheduling (e.g., user/system collects timetable constraints from participants, user/system selects meeting timeslot). Means-based variability may expand the ways meetings can be scheduled, for example, by adding a "meeting scheduling by decree" option where the initiator sets the time and expects participants to re-arrange their schedules accordingly.

Ends-based variability change, on the other hand, changes the purpose of the system itself. For instance, the meeting scheduler needs to be turned into a project management software system, or an office management toolbox. In this case, care needs to be exercised in managing scarce resources (e.g., rooms, people's time). Desai et al. [52] offers a promising direction for research on this form of variability change. Along a different path, Rommes and America [53] proposes a scenario-based approach to creating a product line architecture that does take into account possible long-term changes. through the use of strategic scenarios.

Modularity is another fundamental trait of evolvable software systems. Modularity has been researched throughly since the early 70s. A system is highly modular if it consists of components that have high (internal) cohesion and low (external) coupling. A highly modular system can have some of its components change with low impact on other components. Interestingly, Biology has also studied how coupling affects evolution. In particular, organisms in nature continuously co-evolve both with other organisms and with a changing abiotic environment. In this setting, the ability of one species to evolve is bounded by the characteristics of other species that it depends on. Accordingly, Kauffman [54] introduces the NKC model, named after the three main components that determine the behaviors of species' interaction with one another. According to the model, the co-evolution of a system and its environment is the equilibrium of external coupling and internal coupling. [55] presents a very preliminary

attempt to use this model to account for the co-evolution of software systems along with their environment.

Modularity and variability are clearly key principles underlying the ability of a species to evolve. It would be interesting to explore other principles that underlie evolvability.

There are deeper research issues where advances will have a major influence on solutions for the problem-at-hand. We mention three such issues:

Science of design. According to H. Simon's vision [56], a theory of design that encompasses at least three ingredients: (a) the purpose of an artifact, (b) the space of alternative designs, (c) the criteria for evaluating alternatives. Design artifacts that come with these ingredients will obviously be easier to evolve.

Model evolution. Models will be an important (perhaps the) vehicle for dealing with requirements evolution. Unfortunately, the state-of-the-art in modeling is such that models become obsolete very quickly, as their subject matter evolves. In Physics and other sciences, models of physical phenomena do not need to evolve because they capture invariants (immutable laws).

We either need here a different level of abstraction for modeling worlds of interest to design (usually technical, social and intentional), so that they capture invariants of the subject matter. Alternatively, we need techniques and infrastructures for model evolution as their subject matter changes.

Evolutionary design.[2] Extrapolating from Darwin's theory of evolution where design happens with no designer [57], we could think of mechanisms through which software evolves without any master purpose or master designer. An example of non-directed design is the Eclipse platform (eclipse.org). Rather than one centrally directed, purpose-driven technology, Eclipse has evolved into an ecology supporting multiple components, projects and people, leveraging the advantages of open-source licences. These software ecologies act as incubators for new projects with diverse characteristics. It would be fruitful to understand better the evolutionary processes taking place in these ecologies and invent other mechanisms for software evolution that do not involve a single master designer (also known as intelligent design in some places ...) This is in sharp contrast to Simon's vision. At the same time, this is an equally compelling one.

4 Monitoring Requirements

Requirement monitoring aims to track a system's runtime behavior so as to detect deviations from its requirement specification. Fickas and Feather's work ([58, 39]) presents a run-time technique for monitoring requirements satisfaction. This technique identifies requirements, assumptions and remedies. If an assumption is violated, the associated requirement is denied, and the associated remedies are executed. The approach uses a Formal Language for Expressing Assumptions (FLEA) to monitor and alert the user of any requirement violations.

[2] ... or, "Darwin's dangerous idea" [57].

Along similar lines, Robinson has proposed a requirements-monitoring framework named ReqMon [59]. In this framework, requirements are represented in the goal-oriented requirements modeling language KAOS [60] and through systematic analysis techniques, monitors are extracted that are implemented in commercial business process monitoring software.

We present an alternative approach to requirements monitoring and diagnosis. The main idea of the approach is to use goal models to capture requirements. From these, and on the basis of a number of assumptions, we can automatically derive monitoring specifications and generate diagnoses to recognize system failures. The proposal is based on diagnostic theories developed in AI, notably in Knowledge Representation and AI Planning research [61].

The monitoring component monitors requirements and generates log data at different levels of granularity that can be tuned adaptively depending on diagnostic feedback. The diagnostic component analyzes generated log data and identifies errors corresponding to aberrant system behaviors that lead to the violation of system requirements. When a software system is monitored with low granularity, the satisfaction of high level requirements is monitored. In this case, the generated log data are incomplete and many possible diagnoses can be inferred. The diagnostic component identifies the ones that represent root causes.

Software requirements models may be available from design-time, generated during requirements analysis, or they may be reverse engineered from source code using requirements recovery techniques (for example, Yu et al. [29]). We assume that bi-directional traceability links are provided, linking source code to the requirements they implement.

4.1 Preliminaries

Goal models have been used in Requirement Engineering (RE) to model and analyze stakeholder objectives [60]. Functional requirements are represented as hard goals, while non-functional requirements are represented as soft goals [62]. A goal model is a graph structure, where a goal can be AND- or OR- decomposed into subgoals and/or tasks. Means-ends links further decompose leaf level goals to tasks ("actions") that can be performed to fulfill them. At the source code level, tasks are implemented by simple procedures or composite components that are treated as black boxes for the purposes of monitoring and diagnosis. This allows a software system to be monitored at different levels of abstraction.

Following [63], if goal G is AND/OR decomposed into subgoals $G_1, \ldots, G_n$, then all/at-least-one of the subgoals must be satisfied for G to be satisfied. Apart from decomposition links, hard goals and tasks can be related to each other through MAKE(++) and BREAK(--) contribution links. If a MAKE (or a BREAK) link leads from goal G_1 to goal G_2, G_1 and G_2 share the same (or inversed) satisfaction/denial labels.

As an extension, we associate goals and tasks with preconditions and postconditions (hereafter *effects*, to be consistent with AI terminology) and monitoring switches. Preconditions and effects are propositional formulae, in Conjunctive

Normal Form (CNF), whose truth values are monitored and analyzed during diagnostic reasoning. Monitoring switches can be switched on/off to indicate whether satisfaction of the requirements corresponds to the goals/tasks is to be monitored at run time.

The propositional satisfiability (SAT) problem is concerned with determining whether there exists a truth assignment to variables of a propositional formula that makes the formula true. If such a truth assignment exists, the formula is said to be satisfiable. A SAT solver is any procedure that determines whether a propositional formula is satisfiable, and identifies the satisfying assignments of variables if it is.

The earliest and most prominent SAT algorithm is DPLL (Davis-Putnam-Logemann-Loveland) [64]. Even though the SAT problem is inherently intractable, there have been many improvements to SAT algorithms in recent years. Chaff ([65]), BerkMin ([66]) and Siege ([67]) are among the fastest SAT solvers available today. Our work uses SAT4J ([68]), an efficient SAT solver that inherits a number of features from Chaff.

4.2 Framework Overview

Satisfaction of a software system's requirements can be monitored at different levels of granularity. Selecting a level involves a tradeoff between monitoring overhead and diagnostic precision. Lower levels of granularity monitor leaf level goals and tasks. As a result, more complete log data are generated, leading to more precise diagnoses. The disadvantage of fine-grained monitoring is high overhead and the possible degradation of system performance. Higher levels of granularity monitor higher level goals. Consequently, less complete log data are generated, leading to less precise diagnoses. The advantage is reduced monitoring overhead and improved system performance.

We provide for adaptive monitoring at different levels of granularity by associating monitoring switches with goals and tasks in a goal model. When these switches are turned on, satisfaction of the corresponding goals/tasks is monitored at run time. The framework adaptively selects a monitoring level by turning these switches on and off, in response to diagnostic feedback. Monitored goals/tasks need to be associated with preconditions and effects whose truth values are monitored and are analyzed during diagnostic reasoning. Preconditions and effects may also be specified for goals/tasks that are not monitored. This allows for more precise diagnoses by constraining the search space.

Figure 1 provides an overview of our monitoring and diagnostic framework. The input to the framework is the monitored program's source code, its corresponding goal model, and traceability links. From the input goal model, the parser component obtains goal/task relationships, goals and tasks to be monitored, and their preconditions and effects. The parser then feeds this data to the instrumentation and SAT encoder components in the monitoring and diagnostic layers respectively.

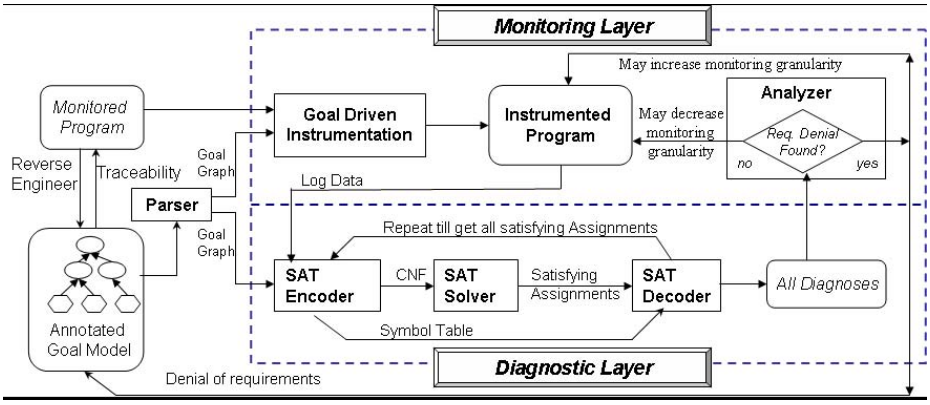

Fig. 1. Framework Overview

In the monitoring layer, the instrumentation component inserts software probes into the monitored program at the appropriate places. At run time, the instrumented program generates log data that contains program execution traces and values of preconditions and effects for monitored goals and tasks. Offline, in the diagnostic layer, the SAT encoder component transforms the goal model and log data into a propositional formula in CNF which is satisfied if and only if there is a diagnosis. A diagnosis specifies for each goal and task whether or not it is fully denied. A symbol table records the mapping between propositional literals and diagnosis instances. The SAT solver finds one possible satisfying assignment, which the SAT decoder translates into a possible diagnosis. The SAT solver can be repeatedly invoked to find all truth assignments that correspond to all possible diagnoses.

The analyzer analyzes the returned diagnoses, searching for denials of system requirements is found. If denials of system requirements are found, they are traced back to the source code to identify the problematic components. The diagnosis analyzer may then increase monitoring granularity by switching on monitoring switches for subgoals of a denied parent goal. When this is done, subsequent executions of the instrumented program generate more complete log data. More complete log data means fewer and more precise diagnoses, due to a larger SAT search space with added constraints. If no system requirements are denied, monitoring granularity may also be decreased to monitor fewer (thus higher level) goals in order to reduce monitoring overhead. The steps described above constitute one execution session and may be repeated.

4.3 Formal Foundations

This section presents an overview of the theoretical foundations of our framework. The theories underlying our diagnostic component (presented in section 4.2) are adaptations of the theoretical diagnostic frameworks proposed in [69,

70, 61]. Interested readers can refer to [2] for a complete and detailed account of the presented framework.

Log Data. Log data consists of a sequence of log instances, each associated with a specific timestep t. A log instance is either the observed truth value of a domain literal, or an occurrence of a particular task. We introduce predicate $occ_a(a_i, t)$ to specify occurrence of task a_i at timestep t. For example, if literal p is true at timestep 1, task a is executed at timestep 2, and literal q is false at timestep 3, their respective log instances are: $p(1)$, $occ_a(a, 2)$, and $\neg q(3)$.

Successful execution of tasks in an appropriate order leads to satisfaction of the root goal. A goal is satisfied in some execution secession s if and only if all the tasks under its decomposition are successfully executed in s. Goal satisfaction or denial may vary from session to session. The logical timestep t is incremented by 1 each time a new batch of monitored data arrives and is reset to 1 when a new session starts.

We say a goal has occurred in s if and only if all the tasks in its decomposition have occurred in s. Goal occurrences are not directly observable from the log data. Instead, our diagnostic component infers goal occurrence from task occurrences recorded in the log. Two timesteps, t_1 and t_2, are associated with goal occurrences, representing the timesteps of the first and the last executed task in the goal's decomposition in s. We introduce predicate $occ_g(g_i, t_1, t_2)$ to specify occurrences of goals g_i that start and end at timesteps t_1 and t_2 respectively. For example, suppose goal g is decomposed into tasks a_1 and a_2, and we have in the log data $occ_a(a_1, 4)$, $occ_a(a_2, 7)$ indicating that tasks a_1 and a_2 have occurred at timesteps 4 and 7 respectively. Then $occ_g(g, 4, 7)$ is inferred to indicate that g's occurrence started and ended at timesteps 4 and 7.

Theories of Diagnosis. The diagnostic component analyzes generated log data and infers satisfaction/denial labels for all the goals and tasks in a goal model. This diagnostic reasoning process involves two steps: (1), inferring satisfaction/denial labels for goals/tasks that are monitored; and (2), propagating these satisfaction/denial labels to the rest of the goal model. Note that if a goal/task is not monitored, but is associated with a precondition and an effect whose truth values are recorded in the log or can be inferred from it, then its satisfaction/denial is also inferred from step 1.

Intuitively, a goal g can be denied in one of three ways: (1) g itself can be denied, if it is monitored or if the truth values of its precondition and effect are known; or (2) one of g's children or parents is denied and the deniability is propagated to g through AND/OR decomposition links; or (3) one of the goals/tasks that are linked to g through MAKE(++)/BREAK(--) contribution links is denied/satisfied, in which case the denial label is propagated to g. As with goals, tasks get their denial labels if they themselves are denied, or if their parents are denied and denial labels are propagated down to them.

We reduced the problem of searching for a diagnosis to that of the satisfiability of a propositional formula Φ, where Φ is the conjunction of the following axioms:

(1) axioms for reasoning with goal/task denials (step 1); and (2) axioms for propagating inferred goal/task denials to the rest of the goal model (step 2).

Axiomatization of Deniability. The denial of goals and tasks is formulated in terms of the truth values of the predicates representing their occurrences, preconditions and effects. We introduce a distinct predicate FD to express full evidence of goal and task denial at a certain timestep or during a specific session. FD predicates take two parameters: the first parameter is either a goal or a task specified in the goal model, and the second parameter is either a timestep or a session id. For example, predicates $FD(g_1, 5)$ and $FD(a_1, s_1)$ indicate goal g_1 and task a_1 are denied at timestep 5 and session s_1 respectively.

Intuitively, if a task's precondition is true and the task occurred at timestep t, and if its effect holds at the subsequent timestep $t + 1$, then the task is not denied at timestep $t + 1$. Two scenarios describe task denial: (1)[3] if the task's precondition is false at timestep t, but the task still occurred at t; or (2) if the task occurred at timestep t, but its effect is false at the subsequent timestep $t + 1$. Task denial axioms are generated for tasks to capture both of these cases.

We illustrate task denial axioms using the following example. Consider a task a with precondition p and effect q. If the monitoring component generates one of the following two log data for a, task a's denial is inferred:

Log data 1: $\neg p(1); occ_a(a, 1)$

Log data 2: $p(1); occ_a(a, 1); \neg q(2)$

The first log data corresponds to the first task failure scenario: a's precondition p was false at timestep 1, but a still occurred at 1. The second log data corresponds to the second failure scenario: a's precondition was true and a occurred at timestep 1, but its effect q was false at the subsequent timestep 2. The diagnostic component infers $FD(a, 2)$ in both of these cases, indicating that task a has failed at timestep 2.

These failure scenarios also apply to goals. Recall that goal occurrences are indexed with two timesteps t_1 and t_2 that correspond to the occurrence timesteps of the first and last executed tasks under goal's decomposition. A goal g with precondition p and effect q is denied if and only if (1) goal occurrence started at t_1 when p is false; or (2) after goal occurrence finished at $t_2 + 1$, q is false.

For instance, if g is decomposed to tasks a_1 and a_2, the following sample log data correspond to the two failure scenarios for goal g:

Log data 3: $\neg p(1); occ_a(a_1, 1); occ_a(a_2, 2)$

Log data 4: $p(1); occ_a(a_1, 1); occ_a(a_2, 2); \neg q(3)$

From either of the two log data, the diagnostic component infers $occ_g(g, 1, 2)$, indicating that g's occurrence started and ended at timesteps 1 and 2 respectively. Log data 3 and 4 correspond to the first and second goal failure scenarios respectively: p is false when g's occurrence started at timestep 1, and q is false after g's occurrence at timestep 3. In either of these cases, the diagnostic component infers $FD(g, 3)$, indicating that goal g is denied at time step 3.

[3] In many axiomatizations it is assumed that $occ_a(a, t) \rightarrow p(t)$.

We say a goal or a task is denied during an execution session s if the goal/task is denied at some timestep t within s. Returning to the above examples, if $FD(a, 2)$ and $FD(g, 3)$ are inferred, and if timesteps 2 and 3 fall within execution session s_1, the diagnostic component further infers $FD(a, s_1)$ and $FD(g, s_1)$. Inferring goal/task denials for an execution session is useful for efficiently propagating these denial labels to the rest of the goal model.

In the AI literature, propositional literals whose values may vary from timestep to timestep are called *fluents*. A fluent f can take on any arbitrary value at timestep $t + 1$ if it is not mentioned in the effect of a task that is executed at timestep t. Axioms are needed to specify that unaffected fluents retain the same the values from timestep to timestep. An axiom is generated to specify that if the value of a fluent f changes at timestep t, then one of the tasks/goals that has f in its effect must have occurred at $t - 1$ and not have been denied at t. In other words, the truth value of f reminds constant from one timestep to the next, until one of the actions/goals that have f in its effect is executed successfully. For example, consider a task a with effect q, and assume q is not in any other goal's/task's effect. Suppose the log data include: $\neg q(1)$, $occ_a(a, 3)$, and $q(5)$. Then an axiom is generated to infer $\neg q(2)$, $\neg q(3)$,and $q(4)$.

4.4 Axiomatization of a Goal Model

Goal/task denials, once inferred, can be propagated to the rest of the goal graph through AND/OR decomposition links and MAKE/BREAK contribution links. Axioms are generated to describe both label propagation processes.

If a goal g is AND (or OR) decomposed into subgoals $g_1,\ldots, g_n$, and tasks $a_1, \ldots, a_m$, then g is denied in a certain session, s, if and only if at least one (or all) of the subgoals or tasks in its decomposition is (or are) denied in s.

Goals and tasks can be related to each other through various contribution links: $\texttt{++}S$, $\texttt{--}S$, $\texttt{++}D$, $\texttt{--}D$, ++, --. Link ++ and link -- are shorthand for the $\texttt{++}S$ and $\texttt{++}D$, and the $\texttt{--}S$ and $\texttt{--}D$ relationships, respectively, and they represent strong MAKE(++) and BREAK(--) contributions between goals/tasks. Given two goals g_1 and g_2, the link $g_1 \xrightarrow{++S} g_2$ (respectively $g_1 \xrightarrow{--S} g_2$) means that if g_1 is satisfied, then g_2 is satisfied (respectively denied). But if g_1 is denied, we cannot infer denial (or respectively satisfaction) of g_2. The meanings of links $\texttt{++}D$ and $\texttt{--}D$ are similar to those of $\texttt{++}S$ and $\texttt{--}S$. Given two goals g_1 and g_2, the link $g_1 \xrightarrow{++D} g_2$ (respectively $g_1 \xrightarrow{--D} g_2$) means that if g_1 is denied, then g_2 is denied (respectively satisfied). But if g_1 is satisfied, we cannot infer satisfaction (or respectively denial) of g_2.

When contribution links are present, the goal graph may become cyclic and conflicts may arise. We say a conflict holds if we have both $FD(g, s)$ and $\neg FD(g, s)$ in one execution session s. Since it does not make sense, for diagnostic purposes, to have a goal being both denied and satisfied at the same time, conflict tolerance, as in (Sebastiani et al., 2004), is not allowed within our diagnostic framework. In addition, the partial (weaker) contribution links HELP(+) and

HURT(−) are not included between hard goals/tasks because we do not reason with partial evidence for hard goal/task satisfaction and denial.

Diagnosis Defined. In our framework, a diagnosis specifies for each goal/task in the goal model whether or not it is fully denied. More formally, a diagnosis D is a set of FD and $\neg FD$ predicates over all the goals and tasks in the goal graph, such that D union Φ ($D \cup \Phi$) is satisfiable. Each FD or $\neg FD$ predicate in D is either indexed with respect to a timestep or a session. For example, if goal g and task a are both denied at timestep 1 during execution session s_1, the diagnosis for the system would contain $FD(a, 1)$, $FD(a, s_1)$, $FD(g, 1)$, and $FD(g, s_1)$.

Our diagnostic approach is sound and complete, meaning that for any D as defined above, D is a diagnosis if and only if $D \cup \Phi$ is satisfiable. A proof of this soundness and completeness property can be found in (Wang et. al, 2007).

Task level denial is the core or root cause of goal level denial. In addition, if a task is denied at any timestep t during an execution session s, it is denied during s. Therefore, it is more useful, for purposes of root cause analysis, that the diagnostic component infer task level denials during specific sessions. We introduce the concept of core diagnosis to specify for each task in the goal graph whether or not it is fully denied in an execution session. More formally, a core diagnosis (CD) is a set of FD and $\neg FD$ predicates over all the tasks in the goal graph, indexed with respect to a session, such that $CD \cup \Phi$ is satisfiable. Consider the same example where goal g and task a are denied at timestep 1 during the execution session s_1. The core diagnosis for the system would only contain $FD(a, s_1)$, indicating that the root cause of requirement denial during s_1 is the failure of task a.

Inferring all core diagnoses for the software system can present a scalability problem. This is because all the possible combinations of task denials for tasks under a denied goal are returned as possible core diagnoses. Therefore, in the worst-case, the number of core diagnoses is exponential to the size of the goal graph. To address the scalability problem, we introduce the concept of *participating diagnostic components*. These correspond to individual task denial predicates that participate in core diagnoses, without their combinations. A participating diagnostic component, PDC, is an FD predicate over some task in the goal model, indexed with respect to a session, such that $PDC \cup \Phi$ is satisfiable.

In many cases, it may be neither practical nor necessary to find all core diagnoses. In these cases, all participating diagnostic components can be returned. However, it is also important to note that, in other cases, one may want to find all core diagnoses instead of all participating diagnostic components. This is because core diagnoses contain more diagnostic information, such as which tasks can and can not fail together.

Our diagnostic approach is sound and complete, meaning that it finds *all* diagnoses, core diagnoses, and participating diagnostic components for the software system. The theory outlined above has been implemented in terms of four main algorithms: two encoding algorithms for encoding an annotated goal model

into a propositional formula Φ, and two diagnostic algorithms for finding all core diagnoses and all participating diagnostic components.

The difference between the two encoding algorithms lies in whether the algorithm preprocesses the log data when encoding the goal model into Φ. The naive algorithm does not preprocess log data and generates a complete set of axioms for all the timesteps during one execution session. The problem with this is the exponential increase in the size of Φ with the size of a goal model. The second and improved algorithm addresses this problem by preprocessing the log data and only generating necessary axioms for the timesteps that are actually recorded in the log data. As demonstrated in [2], this improved algorithm permits the same diagnostic reasoning process while keeping the growth of the size of Φ polynomial with respect to the size of the goal model.

The results of our framework evaluation (subsection 4.6) show that our approach scales to the size of the goal model, provided the encoding is done with log file preprocessing and the diagnostic component returns all participating diagnostic components instead of all core diagnoses. Interested readers can refer to [2] for a detailed account of algorithms and implementation specifics.

4.5 A Working Example

We use the SquirrelMail [71] case study as an example to illustrate how our framework works. SquirrelMail is an open source email application that consists of 69711 LOC written in PHP. Figure 2 presents a simple, high-level goal graph for SquirrelMail with 4 goals and 7 tasks, shown in ovals and hexagons, respectively.

The SquirrelMail goal model captures the system's functional requirements for sending an email (represented by the root goal g_1). The system first needs to retrieve and load user login page (task a_1), then process the sent mail request (goal g_2), and finally send the email (task a_7). If the email IMAP server is found, SquirrelMail loads the compose page (goal g_3), otherwise, it reports IMAP not

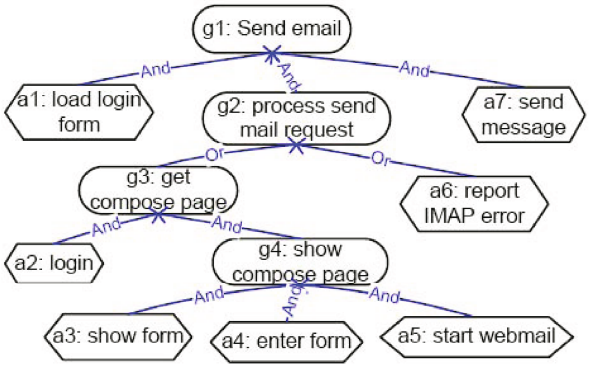

Fig. 2. Squirrel Mail Goal Model

Table 1. Squirrel Mail Annotated Goal Model

Goal/ Task	Monitor switch	Precondition	Effect
a1	on	correctURL entered	login form loaded
a2	on	¬wrongIMAP∧login form loaded	(user logged in ∧ correct pin) ∨ (¬user logged in ∧ ¬correct pin)
a3	off	user logged in	form shown
a4	off	form shown	form entered
a5	off	form entered	webmail started
a6	on	wrongIMAP	error reported
a7	on	webmail started	email sent
g1	off	correct URL entered	email sent ∨ error reported
g2	off	login form loaded ∨ wrongIMAP	webmail started ∨ error reported
g3	off	login form loaded ∧ ¬wrongIMAP	webmail started
g4	on	user logged in	webmail started

found error (task a_6). Goal g_3 (*get compose page*) can be achieved by executing four tasks: a_2 (*login*), a_3 (*show form*), a_4 (*enter form*), and a_5 (*start webmail*).

Table 1 lists the details of each goal/task in the SquirrelMail goal model with its monitoring switch status (column 2), and associated precondition and effect (columns 3 and 4). In this example, the satisfaction of goal $g4$ and tasks a_1, a_2, a_6, and a_7 are monitored.

SquirrelMail's runtime behavior is traced and recorded as log data. Recall that log data contains truth values of literals specified in monitored goals'/tasks' preconditions and effects, as well as the occurrences of all tasks. Each log instance is associated with a timestep t. The following is an example of log data from the SquirrelMail case study:

$$\textit{correct URL entered}(1),\ occ_a(a_1,2),\ \textit{login form loaded}(3),\ \neg \textit{wrongIMAP}(4), occ_a(a_2,5),\ \textit{correct pin}(6), \textit{user logged in}(6),\ occ_a(a_3,7),\ occ_a(a_4,8),\ occ_a(a_5,9), \neg \textit{webmail started}(10),\ occ_a(a_7,11),\ \neg \textit{email sent}(12).$$

The log data contains two errors ($\neg$*webmail started*(10), and $occ_a(a_7, 11)$): (1) the effect of g_4 (web mail started) was false, at timestep 10, after all the tasks under g_4's decomposition (a_3, a_4, and a_5) were executed; and (2) task a_7 (*send message*) occurred at timestep 11 when its precondition webmail started was false at timestep 10. The diagnostic component analyzes the log data and infers that goal g_4 and the task a_7 are denied during execution session s. The diagnostic component further infers that if g_4 is denied in s, at least one of g_4's subtasks, a_3, a_4, and a_5, must have been denied in s. The following seven core diagnoses are returned to capture all possible task denials for a_3, a_4, and a_5:

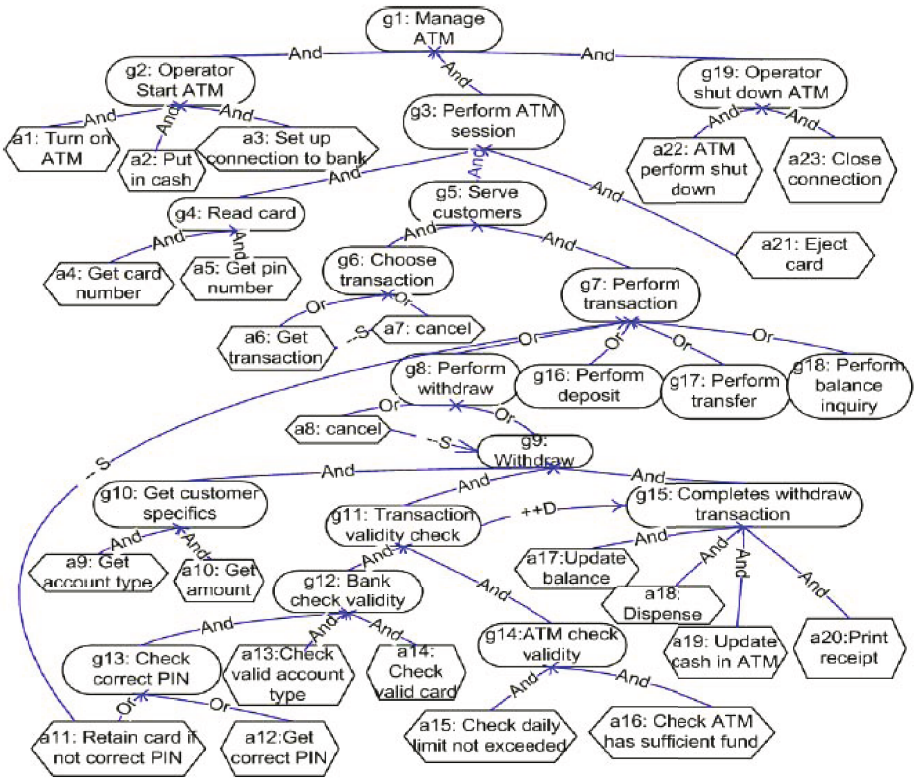

Fig. 3. Partial ATM Goal Model

Core Diagnosis 1: $FD(a_3, s); FD(a_7, s)$
Core Diagnosis 2: $FD(a_4, s); FD(a_7, s)$ *Core Diagnosis 3:* $FD(a_5, s); FD(a_7, s)$
Core Diagnosis 4: $FD(a_3, s); FD(a_4, s); FD(a_7, s)$
Core Diagnosis 5: $FD(a_3, s); FD(a_5, s); FD(a_7, s)$
Core Diagnosis 6: $FD(a_4, s); FD(a_5, s); FD(a_7, s)$
Core Diagnosis 7: $FD(a_3, s); FD(a_4, s); FD(a_5, s); FD(a_7, s)$

Instead of finding all core diagnoses, we can configure the diagnostic component to find all participating diagnostic components. The following 4 participating diagnostic components are returned to capture individual task denials:

Participating Diagnostic Component 1: $FD(a_3, s)$
Participating Diagnostic Component 2: $FD(a_4, s)$
Participating Diagnostic Component 2: $FD(a_5, s)$
Participating Diagnostic Component 3: $FD(a_7, s)$

4.6 Experimental Evaluation

In this section, we report on the performance and scalability of our framework and discuss its limitations. We applied our framework to a medium-size public domain software system, an ATM (Automated Teller Machine) simulation case study, to evaluate the correctness and performance of our framework. We show that our solution can scale up to the goal model size and can be applied to industrial software applications with medium-sized requirements.

Framework Scalability. The ATM simulation case study is an illustration of OO design used in a software development class at Gordon College [72]. The application simulates an ATM performing customers' *withdraw, deposit, transfer* and *balance inquiry* transactions. The source code contains 36 Java Classes with 5000 LOC, which we reverse engineered to its requirements to obtain a goal model with 37 goals and 51 tasks. We show a partial goal graph with 18 goals and 22 tasks in Figure 3.

We conducted two sets of experiments. The first set contains five experiments with different levels of monitoring granularity, all applied to the goal model shown in Figure 3. This allows us to access the tradeoff between monitoring granularity and diagnostic precision. The second set reports 20 experiments on 20 progressively larger goal models containing 50 to 1000 goals and tasks. We obtain these larger goal models by cloning the ATM goal graph to itself. The second set of experiments shows that our diagnostic framework scales to the size of the relevant goal model, provided the encoding is done with log preprocessing and the diagnostic component returns all participating diagnostic components.

The first set of experiments contains 5 runs. We gradually increased monitoring granularity from monitoring only the root goal to monitoring all leaf level tasks. For each experiment, we recorded: (1) numbers of generated literals and clauses in the SAT propositional formula Φ; (2) the number of participating diagnostic components returned; and (3) the average time taken, in seconds, to find one diagnostic component. When the number of monitored goals/tasks was increased from 1 to 11, the number of returned participating diagnostic components decreased from 19 and 1, and the average time taken to find one diagnostic component increased from 0.053 to 0.390 second.

These experiments showed that diagnostic precision is inversely proportional to monitoring granularity. When monitoring granularity increases, monitoring overhead, SAT search space, and average time needed to find a single participating diagnostic component all increase. The benefit of monitoring at a high level of monitoring granularity is that we are able to infer fewer participating diagnostic components identifying a smaller set of possible faulty components. The reverse is true when monitoring granularity decreases: we have less overhead, but the number of participating diagnostic components increases if the system is behaving abnormally. When the system is running correctly (no requirements are denied, and no faulty component is returned), minimal monitoring is advisable.

The second set of experiments, on 20 progressively larger goal models (containing from 50 to 1000 goals and tasks) allows us to evaluate the scalability of the diagnostic component. We injected one error in one of the tasks. Each

of the experiments was performed with complete (task level) monitoring. Each therefore returned only a single diagnostic component. In addition, all experiments used the encoding algorithm that preprocesses log data. This was done to ensure scalability. For each experiment, we recorded: (1) time taken to encode the goal model into the SAT propositional formula Φ; (2) time taken by the SAT solver to solve Φ plus the time taken to decode the SAT result into a diagnostic component; and (3) the sum of the time periods recorded in (1) and (2), giving the total time taken to find the participating diagnostic component.

Experimental results show that, as the number of goals/tasks increased from 50 to 1000, the number of literals and clauses generated in Φ increased from 81 to 1525 and from 207 to 4083 respectively. As a result, the total time taken to find the participating diagnostic component increased from 0.469 to 3.444 seconds. This second set of experiments shows that the diagnostic component scales to the size of the goal model, provided the encoding is done with log preprocessing and the diagnostic component returns all participating diagnostic components. Our approach can therefore be applied to industrial software applications with medium-sized requirement graphs.

Framework Limitations. Firstly, our approach assumes the correct specification of the goal model, as well as the preconditions and effects for goals and tasks. Errors may be introduced if specified preconditions and effects do not completely or correctly capture the software system's dynamics. Detecting and dealing with discrepancies between a system's implementation and its goal model are beyond the scope of our work. We accordingly, assume that both the goal model and its associated preconditions and effects are correctly implemented by the application source code.

Secondly, the reasoning capability of our diagnostic component is limited by the expressive power of propositional logic and the reasoning power of SAT solvers. Propositional logic and SAT solvers express and reason using variables with discrete values, which typically are Boolean variables that are either true or false. As a result, our diagnostic component cannot easily deal with application domains with continuous values.

Lastly, the reasoning power of our framework is also limited by the expressiveness of our goal modeling language. Goal models cannot express temporal relations. Neither can they explicitly express the orderings of goals/tasks, or the number of times goals/tasks must be executed. Therefore, our framework cannot recognize temporal relations such as event patterns.

5 Conclusions

We have discussed requirements evolution as a research problem that has received little attention until now, but will receive much attention in the future. Our discussion included a review of past research, a speculative glimpse into the future, and a more detailed look at on-going research on monitoring and diagnosing software systems.

References

[1] Lubars, M., Potts, C., Richter, C.: A review of the state-of-practice in requirements modelling. In: Intl. Symp. on Requirements Engineering, San Diego, CA (January 1993)

[2] Wang, Y., McIlraith, S., Yu, Y., Mylopoulos, J.: An automated approach to monitoring and diagnosing requirements. In: International Conference on Automated Software Engineering (ASE 2007), Atlanta, GA (October 2007)

[3] Lehman, M.: On understanding laws, evolution, and conservation in the large-program life cycle. Journal of Systems and Software 1, 213–221 (1980)

[4] Brooks, F.: The mythical man-month. Addison-Wesley, Reading (1975)

[5] Fernandez-Ramil, J., Perry, D., Madhavji, N.H. (eds.): Software Evolution and Feedback: Theory and Practice, 1st edn. Wiley, Chichester (2006)

[6] Gîrba, T., Ducasse, S.: Modeling history to analyze software evolution. Journal of Software Maintenance and Evolution: Research and Practice 18(3), 207–236 (2006)

[7] Xing, Z., Stroulia, E.: UMLDiff: an algorithm for objectoriented design di erencing. In: Intl. Conf. on Automated Software Engineering, Long Beach, CA, USA, pp. 54–65 (2005)

[8] Basili, V.R., Weiss, D.M.: Evaluation of a software requirements document by analysis of change data. In: Intl. Conf. on Software Engineering, San Diego, USA, pp. 314–323 (1981)

[9] Basili, V.R., Perricone, B.T.: Software errors and complexity: An empirical investigation. Commun. ACM 27(1), 42–52 (1984)

[10] Harker, S.D.P., Eason, K.D., Dobson, J.E.: The change and evolution of requirements as a challenge to the practice of software engineering. In: IEEE International Symposium on Requirements Engineering, pp. 266–272 (1993)

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

[12] Lientz, B.P., Swanson, B.E.: Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations. Addison-Wesley, Reading (1980)

[13] Rowe, D., Leaney, J., Lowe, D.: De ning systems evolvability - a taxonomy of change. In: International Conference and Workshop: Engineering of Computer-Based Systems, Maale Hachamisha, Israel, p. 45+ (1998)

[14] Vicente, K.J.: Ecological interface design: Progress and challenges. Human Factors 44, 62–78 (2002)

[15] Favre, J.-M.: Meta-model and model co-evolution within the 3d software space. In: Intl. Workshop on Evolution of Large-scale Industrial Software Applications at ICSM, Amsterdam (September 2003)

[16] Swanson, B.E.: The dimensions of maintenance. In: Intl. Conf. on Software Engineering, San Francisco, California, pp. 492–497 (1976)

[17] Svensson, H., Host, M.: Introducing an agile process in a software maintenance and evolution organization. In: European Conference on Software Maintenance and Reengineering, Manchester, UK, March 2005, pp. 256–264 (2005)

[18] Nelson, K.M., Nelson, H.J., Ghods, M.: Technology exibility: conceptualization, validation, and measurement. In: International Conference on System Sciences, Hawaii, vol. 3, pp. 76–87 (1997)

[19] Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G.: Towards a taxonomy of software change. Journal of Software Maintenance and Evolution: Research and Practice 17(5), 309–332 (2005)

[20] Chapin, N., Hale, J.E., Fernandez-Ramil, J., Tan, W.-G.: Types of software evolution and software maintenance. Journal of Software Maintenance and Evolution: Research and Practice 13(1), 3–30 (2001)
[21] Felici, M.: Taxonomy of evolution and dependability. In: Proceedings of the Second International Workshop on Unanticipated Software Evolution, USE 2003, Warsaw, Poland, April 2003, pp. 95–104 (2003)
[22] Felici, M.: Observational Models of Requirements Evolution. PhD thesis, University of Edinburgh (2004),
http://homepages.inf.ed.ac.uk/mfelici/doc/IP040037.pdf
[23] Sommerville, I., Sawyer, P.: Requirements Engineering: A Good Practice Guide. John Wiley & Sons, New York (1997)
[24] Lam, W., Loomes, M.: Requirements evolution in the midst of environmental change: A managed approach. In: Euromicro. Conf. on Software Maintainance and Reengineering, Florence, Italy, March 1998, pp. 121–127 (1998)
[25] Stark, G., Skillicorn, A., Ameele, R.: An examination of the e ects of requirements changes on software releases. Crosstalk: Journal of Defence Software Engineering, 11–16 (December 1998)
[26] Lormans, M., van Dijk, H., van Deursen, A., Nocker, E., de Zeeuw, A.: Managing evolving requirements in an outsourcing context: an industrial experience report. In: International Workshop on Principles of Software Evolution, pp. 149–158 (2004)
[27] Wiegers, K.E.: Automating requirements management. Software Devel- opment Magazine 7(7) (July 1999)
[28] Roshandel, R., Van Der Hoek, A., Mikic-Rakic, M., Medvidovic, N.: Mae – a system model and environment for managing architectural evolution. ACM Trans. Softw. Eng. Methodol. 13(2), 240–276 (2004)
[29] Yu, Y., Wang, Y., Mylopoulos, J., Liaskos, S., Lapouchnian, A.: Reverse engineering goal models from legacy code. In: International Conference on Requirements Engineering, Paris, pp. 363–372 (September 2005)
[30] Niu, N., Easterbrook, S., Sabetzadeh, M.: A categorytheoretic approach to syntactic software merging. In: 21st IEEE International Conference on Software Maintenance (ICSM 2005), Budapest, Hungary (September 2005)
[31] Berry, D.M., Cheng, B.H.C., Zhang, J.: The four levels of requirements engineering for and in dynamic adaptive systems. In: International Workshop on Requirements Engineering: Foundation for Software Quality, Porto, Portugal (June 2005)
[32] Liaskos, S.: Acquiring and Reasoning about Variability in Goal Models. PhD thesis, University of Toronto (2008)
[33] Jureta, I., Faulkner, S., Thiran, P.: Dynamic requirements specification for adaptable and open service-oriented systems. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 270–282. Springer, Heidelberg (2007)
[34] Etien, A., Salinesi, C.: Managing requirements in a co-evolution context. In: 13th IEEE International Conference on Requirements Engineering, Paris, pp. 125–134 (September 2005)
[35] Boehm, B.: Some future trends and implications for systems and software engineering processes. Systems Engineering 9(1), 1–19 (2006)
[36] Yang, Y., Bhuta, J., Boehm, B., Port, D.N.: Value-based processes for cots-based applications. IEEE Software 22(4), 54–62 (2005)
[37] Nuseibeh, B.: Weaving together requirements and architectures. IEEE Computer 34(3), 115–119 (2001)

[38] Gotel, O.C.Z., Finkelstein, C.W.: An analysis of the requirements traceability problem. In: International Conference on Requirements Engineering, pp. 94–101 (1994)
[39] Fickas, S., Feather, M.S.: Requirements monitoring in dynamic environments. In: RE 1995: Proceedings of the Second IEEE International Symposium on Requirements Engineering, Washington, DC, USA (1995)
[40] Feather, M.S., Fickas, S., van Lamsweerde, A., Ponsard, C.: Reconciling system requirements and runtime behaviour. In: Ninth IEEE International Workshop on Software Specification and Design (IWSSD-9), Isobe, JP, pp. 50–59 (1998)
[41] Cleland-Huang, J., Settimi, R., BenKhadra, O., Berezhanskaya, E., Christina, S.: Goal-centric traceability for managing non-functional requirements. In: Intl. Conf. Software Engineering, St. Louis, MO, USA, pp. 362–371 (2005)
[42] Ramesh, B., Jarke, M.: Toward reference models for requirements traceability. IEEE Transactions on Software Engineering 27(1), 58–93 (2001)
[43] Kephart, J., Chess, D.: The vision of autonomic computing. IEEE Computer 36(1) (January 2003)
[44] Cleland-Huang, J., Chang, C.K., Christensen, M.: Event-based traceability for managing evolutionary change. Transactions on Software Engineering 29(9), 796–810 (2003)
[45] Cleland-Huang, J., Settimi, R., Zou, X., Solc, P.: Automated classi cation of non-functional requirements. Requirements Engineering 12(2), 103–120 (2007)
[46] Breaux, T., Antón, A.: Analyzing goal semantics for rights, permissions, and obligations. In: International Requirements Engineering Conference, Paris, France, pp. 177–186 (August 2005)
[47] Siena, A., Maiden, N., Lockerbie, J., Karlsen, K., Perini, A., Susi, A.: Exploring the effectiveness of normative i* modelling: Results from a case study on food chain traceability. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 182–196. Springer, Heidelberg (2008)
[48] Ramadge, P., Wonham, M.: Supervisory control of a class of discreteevent systems. SIAM J. of Control and Optimization 25(1), 206–230 (1987)
[49] Darwin, C.: On the Origin of Species by Means of Natural Selection. Murray, London (1859)
[50] Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Softw. Pract. Exper. 35(8), 705–754 (2005)
[51] Kang, K.C., Kim, S., Lee, J., Kim, K.: Form: A feature-oriented reuse method. Annals of Software Engineering 5, 143–168 (1998)
[52] Desai, N., Chopra, A.K., Singh, M.P.: Amoeba: A methodology for requirements modeling and evolution of crossorganizational business processes. Transactions on Software Engineering and Methodology (submitted, 2008)
[53] Rommes, E., America, P.: A scenario-based method for software product line architecting. In: Käkölä, T., Dueñas, J.C. (eds.) Software Product Lines - Research Issues in Engineering and Management, Berlin, pp. 3–52 (2006)
[54] Kau, S.A.: The Origins of Order: Self-Organization and Selection in Evolution. Oxford University Press, Oxford (1993)
[55] Su, N., Mylopoulos, J.: Evolving organizational information systems with tropos. In: Conference on Advanced Information Systems Engineering (2006)
[56] Simon, H.A.: The Sciences of the Artificial, 3rd edn. MIT Press, Cambridge (1996)
[57] Dennett, D.C.: Darwin's Dangerous Idea: Evolution and the Meanings of Life. Simon & Schuster, New York (1995)
[58] Feather, M.S.: Rapid application of lightweight formal methods for consistency analysis. IEEE Trans. Softw. Eng. 24(11), 948–959 (1998)

[59] Robinson, W.N.: A requirements monitoring framework for enterprise systems. Requirements Engineering Journal 11, 17–41 (2006)
[60] Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Science of Computer Programming 20(1-2), 3–50 (1993)
[61] McIlraith, S.: Explanatory diagnosis: Conjecturing actions to explain observations. In: International Conference on Principles of Knowledge Representation and Reasoning (KR 1998), Trento, Italy, June 1998, pp. 167–179 (1998)
[62] Mylopoulos, J., Chung, L., Nixon, B.: Representing and using non-functional requirements: A process-oriented approach. IEEE Transactions on Software Engineering 18(6), 483–497 (1992)
[63] Giorgini, P., Mylopoulos, J., Nicchiarelli, E., Sebastiani, R.: Reasoning with goal models. In: Spaccapietra, S., March, S.T., Kambayashi, Y. (eds.) ER 2002. LNCS, vol. 2503, pp. 167–181. Springer, Heidelberg (2002)
[64] Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Journal of ACM 5, 394–397 (1962)
[65] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient sat solver. In: Design Automation Conference, Las Vegas, pp. 530–535 (June 2001)
[66] Goldberg, E., Novikov, Y.: Berkmin: A fast and robust sat-solver. In: Conference on Design, Automation and Test in Europe (DATE), Paris, pp. 142–149 (March 2002)
[67] Ryan, L.: Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University (2004)
[68] Le Berre, D.: A satisfiability library for java (2007), `http://www.sat4j.org/`
[69] Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence 32(1), 57–95 (1987)
[70] De Kleer, J., Mackworth, A.K., Reiter, R.: Characterizing diagnoses and systems. Artificial Intelligence 56(2-3), 197–222 (1992)
[71] Castello, R.: Squirrel mail (2007), `http://www.squirrelmail.org/`
[72] Bjork, R.: An example of object-oriented design: an atm simulation (2007), http://www.cs.gordon.edu/courses/cs211/ATMExample/index.html/

APPENDIX B – Security Requirements Engineering for Evolving Software Systems: A Survey

Security Requirements Engineering for Evolving Software Systems: a Survey

Abstract

Long-lived software systems often undergo numerous evolutions over an extended period of time. Continuous evolution of these systems is inevitable as they need to continue to satisfy changing business needs, new regulations and standards, and introduction of novel technologies in their operating environment. Evolution of systems may involve changes that add, remove, or modify features; or that migrate the system from one operating platform to another. These changes may result in requirements that were satisfied in a previous release of an application not being satisfied in its updated version. When evolutionary changes violate security requirements, a system may be left vulnerable to attacks. In this paper we review current approaches to security requirements engineering and conclude that they lack explicit support for managing the effects of software evolution. We then suggest that a cross fertilisation of the areas of software evolution and security engineering would address the problem of maintaining compliance to security requirements of software systems as they evolve. We conclude the paper with a research agenda that highlights research issues that may need to be addressed.

1. Introduction

Software evolution refers to the process of continually updating software systems in response to changes in their operating environment and their requirements [90, 91]. These changes are often driven by business needs, regulations, and standards that a software application is required to continue to satisfy or adapt to [18, 84]. The changes to a software system may involve adding new features, removing, modifying existing features [20, 78], redesigning the system for migration to a new platform, or integration with other applications. Such changes may result in requirements that were satisfied in a previous release of an application being violated in its updated version [45, 46].

Security requirements engineering deals with the protection of assets from potential threats that may lead to harm [52]. This paper observes that current approaches to security requirements engineering have limited capability for preserving security properties that may be violated as a result of software evolution. In supporting this argument we review the state-of-the-art in both literatures of software evolution and security engineering, noting research challenges.

In illustrating the need for security requirements engineering approaches to support software evolution, we consider how the introduction of a government regulation which states that only employees with valid work permits are allowed to work may affect a standalone payroll system. One way to enforce this regulation could be to introduce a feature that allows a central immigration control system to access employee database records in the payroll system. Such a change, however, may require migrating the payroll system to a platform that supports public network access (such as the Internet) where it can communicate with remote applications. Allowing the immigration control application access to the payroll implies that immigration officers now have access to private employee data which were only available with the consent from the individual employees previously. Such evolution of the payroll system has violated the confidentiality requirements of employees.

The example illustrates the need for security requirements engineering to incorporate requirements evolution. We suggest that one way to address the problem of violation of security requirements as result of evolution is a cross fertilisation of approaches to managing software evolution with security requirements engineering. We hope that the cross fertilisation leads to an ideal approach to *security requirements engineering for evolving systems*.

However, there are a number of challenges that will have to be addressed in order to make such cross fertilisation a reality. The theme of these challenges is how to design software systems so that they are both secure and evolvable. Current research in software evolution does not explicitly address security issues and approaches to security requirements engineering do not provide systematic means to

addressing software evolution concerns. Hence addressing the challenge of secure software evolution will, inevitably, involve identifying promising approaches in both research fields and then finding some way of integrating them. We anticipate that such cross fertilisation is non-trivial as it has to strike a balance between security and evolution of software systems. Meeting these challenges will involve inventing a process for security requirements analysis in long-lived systems and a methodology for evolutionary requirements.

The challenge of achieving security requirements engineering for evolving software systems is made harder by the fact that achieving software systems that both evolvable and secure can be conflicting goals [111]. One of the key characteristics of software evolution is that in response to new requirements, new features may be added to existing systems. This mandates composition of the existing feature set with new features. However, feature composition is non-monotonic [146]; that is, properties that were true of an existing system before combination with a new feature, are not guaranteed to hold after the addition of new functionality.

The context of our work security requirements engineering and for this reason in Section 2 we attempt to define what software evolution might mean from a requirements engineering perspective. We do this by framing evolution research into a requirements engineering framework. In Section 3 we review the state of the art on approaches to understanding and managing requirements evolution. Section 4 reviews approaches to eliciting and analysing security requirements and Section 5 presents a comparative evaluation of the extent to which security requirements engineering approaches support software evolution. The main objective of this survey is to identify research challenges that need to be addressed and to present a research agenda in order to make security requirements engineering for evolving systems possible. Section 6 discusses these challenges and where possible identifies promising approaches that could be leveraged to address them, from both software evolution and security requirements engineering perspectives. We present a conclusion in Section 7.

2. Software Evolution from a Requirements Engineering Perspective

Software evolution refers to the process of developing a software system initially and continually updating it due to change in its stakeholder needs and its operating environment [89-91]. In this section we discuss basic notions of the concept of software evolution (in section 2.1) and examine what software evolution means from a requirements engineering perspective. Our discussion is based on Zave and Jackson's *entailment relation* [155] and we briefly describe this in Subsection 2.2. The entailment relation structures software development problems into three main elements: requirements, specification, and context. Subsections 2.3, 2.4, and 2.5 reviews literature that supports the notions of software evolution as change in context, specification, and context, respectively. Subsection 2.6 concludes the section with a discussion of the implications of a requirements engineering perspective on software evolution.

2.1 Basic Concepts of Software Evolution

Lehman [91], one of the pioneers of software evolution research, identified several trends in software systems that seemed define intrinsically the nature of software systems in that they are independent of system stakeholders [24]. This included the observation that with time software systems tend to increase in size and complexity and a result of this increase their maintenance and adaptation becomes more challenging. Although, it is true that software systems increase in size and complexity with time, the view that this is somehow independent of stakeholder intentions is somehow controversial. This is because an increase in software size could be viewed as a reflection of an expansion to the set of stakeholder requirements which may in turn mean an increase in complexity. An empirical study conducted by Barry *et al.* [8] concluded that some of Lehman's hypotheses (called laws of software evolution [88, 91]) on how software system evolve were invalid. . This could be attributed to the fact that these hypotheses are founded on empirical studies of evolution in monolithic systems.

Lehman's theories and laws on software evolution are generic. In this section we evaluate these generic concepts of software evolution from a requirements engineering perspective. More specifically, we examine what software evolution means in terms of Jackson and Zave's entailment relation [155] which describes software in terms of *requirements*, *specification*, and *context*. In the evaluation we review the literature on software evolution and attempt to classify it according to whether it views

software evolution as change in requirements, specification, or context. In doing so we hope to clarify what software evolution means in requirements engineering.

2.2 Jackson and Zave's Entailment Relation

The entailment relation relates three sets of descriptions: *requirements* (R), *domain assumptions* (W), and *specifications* (S). It states that a specification satisfies a requirement given that some assumptions about the behaviour of the context hold (formally, $S, W \vdash R$, where "|-" denotes entailment). A requirement describes a condition or capability that must be met or possessed by a system, in other words, its purpose. Requirements are optative descriptions in that they described how the world would be once the envisioned system is in place. For an electronic stability programme (ESP) feature in a car this could be: 'avoid vehicle skidding when brakes are applied'.

Domain assumptions describe facts about the behaviour of the environment where a system will be deployed. In this paper we use the term *context* to refer to the environment described in domain assumptions. In contrast to requirements, domain descriptions are indicative in that they describe objective truth about the context. In the ESP example this could be: 'applying brakes continuously cause tires to lock', 'tires are mounted on the vehicle's chassis', and 'locked tires lead to vehicle skidding'.

Specifications then describe how the system should behave in order to satisfy the conditions described in R, given that the assumptions described in W hold. The specification for the ESP could be: "if tire lock occurs during braking, apply and release braking pressure at short discrete periodic intervals'.

2.3 Evolution as Change in Context

The operating environment or context of an application plays an important role in its evolution as it is one of the major drivers of evolution. This is especially true for embedded systems [23]. Examples of contextual changes include government regulations [18] (as illustrated by the payroll example in section 1), business process models [63, 137], platforms [43], anomalies observed in the operation of an application resulting from incompleteness of requirements and hardware failures or limitations which were not considered initially [100] and software bugs [148], and inconsistencies between requirements [33, 112, 128, 144].

The design of a software system makes assumptions about the environment in which it will operate [31, 43]. We illustrate this by citing examples from the literature. For example, the migration of a system that was originally designed to run on a Desktop PC to a mobile device has to take into account the limitation of resources in the new platform. The characteristics of the new mobile device platform forces new stringent requirements on the applications use of resources. Another source of change is inconsistencies that arise as features designed independently are composed [127]. Such inconsistencies arise due to the invalidation of assumption that each feature made about the behaviour of the context. For example in a smart home a security feature may require a window to be closed at night while a climate control feature may require the same window to be opened to maintain a cool temperature in the house [81]. Note that the conflict arises because of the two features sharing the same resource and hence is manifested on the context [111]. Inconsistencies between features can lead to system evolution as their resolution may require changing the requirements of each of the features involved in the conflict. In the example of conflicting security and climate control example, a new requirement may be introduced that states priority between the two features and determines which feature should be given control of the window in the event that a conflict occurs.

We have illustrated how changes in context may lead to software evolution and shown that such contextual changes are translated into new requirements that an application has to satisfy in order to remain relevant and effective in its environment [84]. Therefore evolutionary changes in context may eventually be translated into new requirements and hence 'evolution as change in context' results in requirements evolution. It worth noting that an application does not only evolve to satisfy new requirements imposed by changes in context but may also evolve to take advantage of new features available in the context. For example, over the years Microsoft Word has had new functionality introduced due to availability of novel features as the Windows operating system evolved [62].

2.4 Evolution as Change in Specifications

Research in software evolution has traditionally focussed on changes in source code [7, 44, 104, 124, 156] and software architecture [23, 31, 85, 126] as prime variables of system evolution. This has led to techniques such as program refactoring [28, 82, 135] and architectural configuration management systems [126]. In this paper we consider software architectures and code as solutions that are designed to satisfy requirements of an application. Hence we classify them as specifications.

While changes in context may lead to new requirements or to changes in existing requirements, in contrast, evolution of specifications is driven by changes in requirements [45] and as such does not always lead to evolution in requirements. A prime illustration of this point is code refactoring – where the structure of program code may be changed without changing business logic. On the other hand, a change in a requirement often results in a change in business logic [32, 127, 157].

Configuration management systems have been successfully applied to managing evolution at the source code level. Contrary, applying configuration management concepts to the evolution of software architectures leads to many problems as noted by Roshandel *et al.* [126]. In addressing this problem Roshandel *et al.* proposed an approach that combines configuration management and architectural concepts. Although the approach has been validated through its application to a number of projects, there is no evidence that its conceptual basis can be generalised to the evolution of other software artefacts. For example it is not known whether the combination of requirements analysis approaches with configuration management concepts can shed some light on the evolution of requirements. Roshandel et al. idea approach could be further enhanced integrating it with Chung and Sibramanian's [23] approach is based on the ideas that architectural evolution can be achieved by making software architectures themselves evolvable.

2.5 Evolution as Change in Requirements

In recent years, researchers in software evolution have turned their attention to changes in stakeholder needs (expressed as requirements) as one of the drivers of software evolution [29, 58, 131, 157]. Several approaches have been proposed for supporting requirements evolution.

Zowgi and Offen [157] proposed modelling and reasoning about the evolution of requirements using meta level logic for formally capturing intuitive aspects of managing changes to requirements models. The approach involves encoding requirements models as theories and reasoning about changes is achieved by mapping changes between requirements models. A significant limitation of this approach is the overhead of encoding requirements models in logic.

Russo *et al.* [127] proposed an approach to restructuring requirements to facilitate inconsistency detection and change management. The approach was later extended by Garcez et al [29] to combines abductive reasoning and inductive learning for evolving requirements specifications. The aim of Garcez *et al.*'s approach is to preserve goals and requirements during evolution and it is based on the idea of analysis and revision. During analysis, a specification is checked whether it satisfies a given requirement using the concepts of model checking [21, 36, 94, 151]. If the current specification does not satisfy the requirements, diagnosis information is generated which describes on how the specification should be modified in order to satisfy the requirement. During revision the specification is changed so that it satisfies the requirement.

The main feature of the analysis-revision approach is that evolutionary changes are allowed to happen first and their impact on satisfaction of requirements is verified as the next step. This characteristic may not be desirable if the evolutionary changes violate requirements in a manner that causes irreversible damage. The efficiency of the analysis-revision cycle is also heavily dependent on choice of good training examples. A related issue is that the approach does make a provision for validating the automatically generated diagnosis against real-world system properties.

The evolution of requirements expressed in natural language is challenging as it makes it difficult to precisely capture requirements changes. Fabrinni *et al.*'s [32] approach to controlling requirements evolution uses formal concept analysis to enable a systematic and precise verification of consistency among different stages of natural language requirements evolution. Software evolution may also introduce inconsistencies between requirements. Ghose's [45] framework formal approach is aimed at

addressing the problem of requirements inconsistencies resulting from evolution. Similar to Garcez *et al.* [29], Ghose's approach is based on formal default reasoning and belief revision, and is supported by automated tools [46].

This approach has interesting features for addressing requirements evolution in the context of security requirements engineering as it offers a basis for reasoning about the impact of change on the consistency between requirements. However, it is more general and not specific to addressing security issues. Nevertheless, its combination with an analysis and revision approach (such as that of Garcez *et al.*'s) can be useful as a basic building block for an approach to integrating requirements evolution in security engineering.

A framework proposed by Lam and Loomes [84] suggests that one approach to the requirements evolution problem is to have two models: a meta model and a process model. A meta-model captures a set of requirements evolution concepts such as change, impact, and risk. A process model provides a framework for handling the emergence of new or changing requirements. While this framework seems a useful abstraction for approaches to requirements evolution the role of the meta-models is not well motivated. Only capturing the concepts of requirements evolution is not sufficient. It may be more useful if the meta-models provide methods and tools for analysing and eliciting change, determining the impact of the change and its associated risk. These meta activities can be performed through existing change impact analysis approaches [1, 15, 58, 124]. Lemoine and Foisseau [92] also proposed the use of UML meta models for recording the artefacts of the produced as a result of evolution in high assurance systems. Their Meta models do not only capture changes but can also be translated into verification rules that can be used for checking properties of an evolving system such as compatibility between multiple releases. Another technique proposed for validating requirements models during evolution is through simulation [131].

2.6 Implications for a Requirements Engineering Perspective on Software Evolution

In this section we have discussed software evolution from a requirements engineering perspective. The objective of our discussion has been to examine what software evolution means in requirements engineering. Our discussion has focussed on software evolution in light of Jackson and Zave's entailment relation. Through the discussion we have observed that software systems evolve with changing user needs and in their environment. Changes in the context of a system may lead to new requirements or modification of existing requirements.

One the other hand, evolution in specifications does not always result in a corresponding evolution in requirements. This is due to the notion that requirements state stakeholder needs or the problems to be solved, while specifications describe the behaviour of software solutions that could satisfy the requirements. As a result the abstract problem stated as a requirement may remain the same even though its solutions may get progressively refined due to changes in context such as introduction of novel technologies.

We envisage that the observations from our discussion may have important implications for research in software evolution. The main implication concerns approaches to change impact analysis. For example the observation that changing requirements may lead to changing specification could lead to a framework for understanding the impact of changes and traceability of the changes through artefacts in both requirements and specifications.

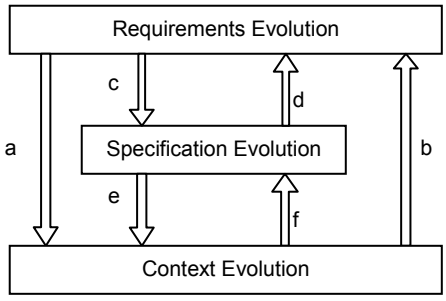

Figure 1. Software Evolution through Entailment Relation

Similarly, such a change impact analysis framework could also be useful for analysing what impact changes in context may have on requirements and specifications. The change impact framework can be validated by doing more research on what the interaction is between the changes in the three elements of the entailment relation as illustrated in Figure 1. The arrows labelled *a* and *b* represents how changes in requirements impact context and how context evolution impact requirements evolution, respectively. Similarly, the arrows labelled *c* and *d* represent the impact of requirements evolution on specification evolution and impact of specification evolution on requirements, respectively. Arrow *e* represent the impact of changes in specification on context, meanwhile arrow *f* represents the impact of changes in context on specifications.. In the next section we review approaches to understanding and managing software evolution.

3. Approaches to Understanding and Managing Software Evolution

Approaches to the study of software evolution can broadly be classified into two categories: explanatory and management [25]. *Explanatory* approaches take a scientific view and are concerned with understanding causes, processes, and effects of software evolution. For example, these approaches study evolution histories of an application in order to understand how the system has changed in response to changes in organisational goals over time [4, 79, 85, 105]. In contrast *management* approaches take an engineering perspective and are concerned with the development of better methods and tools that can be used for managing the effects of software evolution. We review explanatory and process improvement approaches in sections 3.1 and 3.2, respectively.

3.1 Explanatory Approaches

In this section we review exploratory approaches and classify them into two categories based on the type of data they use. The first category use historical data such as changes in source code over a period of time and the second category attempts to understand software evolution by using software trails. We also look at some tools that have been designed to support exploratory approaches.

Using Historical Data to Study Software Evolution: Anton and Potts [4, 5] advocate for the study of the functions offered by a system over its lifetime as a basis for understanding or predicting characteristics of the particular system or similar systems. In realising this idea they proposed an approach, called *functional palaeontology*, to studying the evolution of user-visible features independent of architecture and design intent. The approach is similar to other approaches that study evolution histories [8, 44, 49, 83, 119, 120]. Their approach was motivated by the observation that research in evolution focussed on code-based analysis. They applied the approach to the analysis of evolution in telecommunications features over a 50 year period.

In relation to the entailment relation described in section 2, functional palaeontology studies software evolution in terms of requirements, in contrast to specifications and design. Functional palaeontology has two components: functional morphology and functional evolution. *Functional Morphology* refers to the profile or snapshot of the requirements of an application at a given time. Meanwhile *functional evolution* refers to the patterns of change in the requirements over time. Functional palaeontology was also applied to the study of feature evolution in a word processor [60].

Based on Anton and Potts idea of functional palaeontology [4, 5], Girba and Ducasse [49] proposed Hlsmo – a metamodel in which functional evolution history is modelled as an explicit entity. Hlsmo was motivated by the lack of an explicit meta-model for software evolution analysis. Gall et al. [39], Rysselberghe and Demeyer [129], Wu *et al.* [149] proposed visualisation approaches for understanding software release histories. Although useful, their approaches analyse evolution at the source code level. Using source code analysis to understand evolution is necessary but not sufficient in understanding evolution at the requirements level.

Using Software Trails to Study Evolution: While a majority of explanatory approaches use source code change history for understanding software evolution [51], there are other methods which have been proposed that use a different kind of data. German [44] proposed a method to recovering and analysing the evolution of a software system using software trails. Software trails refers to information behind by contributors to the development process of a software product such as software releases, documentation, version control logs, and websites.

German's approach takes the software trails as input and reconstructs the evolution of an application. Fischer and Gall [35] proposed an approach to analysing feature evolution in software system. The main idea of their approach is to examine hidden dependencies between structurally unrelated features, which over time become coupled. The authors claim that such hidden feature dependencies must be identified as they may be a clear indication of architectural erosion. Architectural erosion refers to any detrimental deviation, with time, of a system's architecture from its original design conception [114].

Tool Support of Explanatory Approaches: A number of tools have been developed to support research in understanding how systems evolve [57]. Hassan and Holt [57] proposed using evolutionary code extractors as tools to help in empirical source code evolution research. Evolutionary code extractors are limited to understanding source code evolution – they cannot be used for understanding other software design artefacts such as specifications and designs. Scalability is also a major challenge in these tools as they have to be able to analyse large and complex code repositories.

3.2 Management Approaches

According to Zave [153] and Chung *et al.* [23] , from a software engineering perspective software evolution is a naturally occurring phenomena. It happens regardless of the actions of designers or requirements analysts and thus can only be managed rather than be controlled. Based on this notion, Zave distinguished research in managing software evolution in terms of prescriptions for evolving systems rather than for evolvable systems. There is an important distinction between *evolving* and *evolvable* systems. The term evolving system refers to the notion that evolution of software systems is a natural process [23, 24, 157]. One the other hand, the natural evolution of a software application may lead to side effects such as architectural erosion and increase in maintenance effort. A common approach to minimising such negative effects is to manage the evolution of a system by designing it in such a way that it is evolvable. We classify management approaches into the following categories: feature engineering and software architectures, continuous architectural evaluation, minimising and accommodating change, change impact analysis, incremental model transformation.

Feature Engineering and Component Architectures: Zave's prescriptions for making software systems evolvable are *feature engineering* and *component architectures*. Feature engineering involves describing all features as if they were independent, understanding all potential feature interaction, deciding on which interactions are desirable and which are not, and adjusting features and composition operators so that features interact only in desirable ways [125, 140, 141, 152]. Component architecture supports feature engineering by providing structural bases on which new functions can be added freely by adding component programs [16, 17, 22, 26, 41, 66, 142].

The main objective of both feature engineering and component architectures is encouraging modular software development. Modularisation seems to be a dominant concept to the management of software evolution and has appeared in different forms such as code clusters [7, 50], feature views [51], components [148] and modules [156]. Based on the observation that managing evolution in a large and complex software is challenging, Glorie *et al.* [50] proposed an approach for splitting a large software repository into smaller repositories. The assumption is that smaller software repositories ease future software evolution.

In re-modularising a repository their approach identifies conceptual commonalities using formal concept analysis and clustering techniques. In validation experiments conducted by the authors, the approach failed as the results of a modularisation were not sensible to domain experts, owing to lack of domain knowledge input into the modularisation process. Although the approach worked for smaller code repositories, it failed to scale-up for larger and more complex source code repositories.

Pena et al. [118] proposed a novel view to software evolution. They view an evolutionary system as being a software product line. The core architecture is the unchanging part of the system and each version of the system may be viewed as a product from the product line. Each product is then described as the core architecture with some additions. However, the assumption that core architecture remains the same in the entire life of an application may not always be true.

Continuous Architectural Evaluation: Del Rosso [31] proposed continuous architectural evaluation as an approach to managing evolution. The approach is based on the premise that software development is

a process of continuous modelling and refinement. Over time the system architecture ages and weakens the system's ability to incorporate new features.

The objective of continuous architecture evaluation is to ensure that the architecture continues to satisfy its requirements. In principle, this approach is similar to Garcez *et al.* [29] approach of combining abductive reasoning and inductive learning to evolve requirements specifications. While Del Rosso's approach focuses on architecture, Garcez *et al.*'s approach is more generic and hence applicable to the evolution of different aspects of software.

Minimising and Accommodating Change: The inherent complexity of software systems increases their susceptibility to fragility due to change induced by unpredictable variations in user needs and technology advances. In addressing this problem Ravichandar *et al*. [123] proposed a capabilities-based approach to constructing complex software systems in such a way that they are tolerant to change. Capabilities are change-tolerant functional abstractions that are foundational to the composition of system functionality derived from user needs [122]. Capabilities are based on the notion that the basic human need for a software system remains the same even though its solution may progressively become more and more refined over time as novel technologies become available.

For example, consider the basic problem of enabling people living in different parts of the world to communication. In ancient time, the initial solution to the problem was to use smoke signalling as a way of relaying information or sending messengers. Then came fixed line telephony and with the introduction of cellular phones came the now popular Short Message Service (SMS). Nowadays instead of sending a messenger, we send an email or SMS. In essence capabilities capture the basic problem to be solved by a software system.

Their idea of capabilities is similar to that of goals that are more tolerant to change compared to low-level requirements. Both approaches are meant to minimize and accommodate change. However, there some differences though. Goal-oriented approaches are top-down as goals are decomposed and refined into requirements that can be implemented [143, 144]. On the other hand, Ravichandar *et al*'s capabilities-based approach [122, 123] is bottom-up as capabilities are derived from requirements. Hence a capabilities-based approach may be used to infer goals from requirements in a similar manner as Van Lamsweed and Willemet's approach to inferring declarative requirements from operational scenarios [145].

Change Impact Analysis: Analysing and understanding the impact of change is one of the problems at the forefront of software evolution research [97, 137]. Soffer's [137] scope analysis approach determines the extent to which changes to one business process affects other business processes. Understand the effects of changes at the level of business processes is useful as such changes are mirrored by evolution of the software systems that support the business processes.

Although Soffer's approach gives requirements analysts an idea of the scope of change, it does not offer a practical method of tracking the impact of changes to the software systems that support the business process. In addressing the change tracking problem Lin et al. [97] proposed capturing requirements changes as a series of atomic changes in specifications and using algorithms to relate changes in requirements to corresponding changes in specifications. A combination of the two approaches has the potential to lead to an approach both for understanding the scope of changes and a way of tracking them as a system evolve. Change impact analysis is a critical component of understanding and managing software evolution. This is evident in the research activities addressing this problem [1, 15, 58, 124].

Incremental Model Transformation: Automated model transformation plays an important role in model-driven system engineering in order to query, derive and manipulate large industrial models. For instance, meta-modeling-based development architectures (including MDA) highly rely on transformations within and between different models and languages. The contribution of model transformation (MT) languages and tools in the overall success of model-driven system development has been reported in many surveys and papers during the recent years [14, 27, 42]. Approaches to model transformation and various solutions addressing the encountered challenges are continuously being explored.

Tool integration based on model transformations is one of the most challenging tasks with high practical relevance. In tool integration, a complex relationship needs to be established and maintained between models conforming to different domains and tools. Similar problems arise in a wide range of circumstances; synchronization involving requirement and design models is an example. The aim of model synchronization is to keep a model of a source language and a model of a target language consistently synchronized while the underlying source model (and sometimes the target also) is evolving. Model synchronization is frequently captured by transformation rules [11]. When the transformation is executed, trace signatures are also generated to establish logical correspondence between source and target models.

Traditionally, model transformation tools support the batch execution of transformation rules, which means that input is always processed "as a whole", and output is always regenerated completely. However, in case of large, complex, and continuously evolving models, batch transformations may not be feasible. To address the issue of model evolution, incremental model transformation: (i) update existing target models based on changes in the source models [121], and (ii) minimize the parts of the source model that need to be reexamined by a transformation when the source model is changed [12]. In the terminology of [27], these aspects are called referred to as target and source incrementality, respectively.

3.3 Summary of Approaches to Software Evolution Management

The approaches discussed in this section focus on understanding and managing software evolution in long-lived systems. Tables 3.1 and 3.2 present summaries of the approaches we have reviewed to understanding and managing evolution, respectively. Worth noting is that a majority of these approaches seem to consider software evolution as change in source code. In table 3.2 the last column represents the level at which each evolution management approach manage evolutionary changes. This can be at the level of requirements (R), specifications (S), or contexts (W).

Table 3.1. Summary of Approaches to Understanding Software Evolution

Explanatory Approach Classification	Type of Data Used	Main Characteristics	Examples Approaches		
Source Code Change History	Source Code	Source code history is used to identify trends in system changes over time.	Empirical Studies	Meta Models	Visualisation
			Anton and Potts [4-6] , Barry *et al*. [8], Smith et al. [136]	Girba and Ducasse [49],	Gall *et al.* [39], Bohner [15], Wu et al. [149]
Software Trails	software releases, system user manuals, version control logs, project websites, Emails.	Uses artefacts generated in support of a software development project, such as emails, to understand source and history of changes.	German [44], Kitchenham *et al*. [80], Kagdi et al. [76].		

Table 3.2. Summary of Approaches to Managing Software Evolution

Management Approach Classification	Main Characteristics	Evolution Management Approaches	Level at Which Change is Managed		
			R	S	W
Feature Engineering	Modularity is used as means to promote ease of evolution and maintenance.	Zave [152, 153]	√	√	√
		Turner et al. [141]		√	
		Ricci [125].		√	
Component Architectures	Propose that software architectures should be design such that they ease of addition and removal of software modules	Bond *et al*. [16]		√	
		Kang *et al*. [77]	√	√	
		Lee *et al*. [87]		√	
		Oreizy *et al*. [115]		√	
		Turner [142]	√	√	
		Zave *et al.* [154]	√	√	√
Continuous Architectural Evaluation	The ability of software architecture to satisfy requirements is continuously verified.	Del Rosso [31]		√	
		Garcez *et al*. [29].	√	√	
Minimising & Accommodating Change	Propose methods and techniques that reduce fragility of software designs as a system evolves.	Ravichandar *et al*. [122, 123]		√	
Change Impact Analysis	Analyse the impact that a change would have on a system design.	Soffer [137],	√		√
		Lin *et al*. [97]		√	
		Ahmad et al. [1]	√	√	
		Bohner [15]		√	
		Hassine et al. [58]	√	√	
		Ren et al. [124].		√	

While is true that changes to a software system are eventually implemented in source code, this limited view of software evolution does not explicitly consider the notion that changes reflected in code actually origination from requirements. Changes at the source code level are at a level too low to enable the understanding of how changes in goals of an organisation are reflected in the source code.

Therefore considering software evolution only at the source code level is insufficient as it does not capture high-level changes such as changes in requirements. Studying evolution at the requirements level could be complementary to the source code level research as it can potentially allow for systematic traceability of how changes at the requirements level are propagated to source code. Most importantly, current approaches to software evolution do not take into account the impact of evolution of security concerns. Kagdi et al. [76] provides a more comprehensive survey and taxonomy of approaches for mining software repositories in the context of software evolution. In the next section we review approaches to security requirements engineering.

4. Security Requirements Engineering

Security is an important characteristic of software systems and it is increasingly considered as a fundamental part of the software development lifecycle. A first step towards this idea is the proposal by Mouratidis *et al.* [107, 109] that security engineering should be an integral part of software engineering. This is based on the notion that an ad hoc integration of security into a software system that has already been developed has a negative effect on its maintainability and security.

In this section we review approaches to security requirements elicitation and analysis. We classify these approaches according to the constructs that they are founded on, namely: goals-based, model-based, problem-based, and process-oriented approaches. Our classification is partly based on previous surveys by Tondel et al. [138], Villarroel et al [147], and Mouratidis and Giorgini [107] and partly by our own understanding of the literature in this area.

4.1 Goal-Based Approaches

Goal-oriented approaches to security engineering focus on identifying threats to satisfaction of goals as the basis for identifying system vulnerabilities. In comparison to low-level requirements, the high-level abstraction of goals implies that they are more stable than low-level requirements. This makes goals less likely to change compared to low-level requirements. However, a limitation resulting from this benefit is that goals may be insufficient for analysing low level security concerns. In this section we review four goal-based approaches: KAOS [143], an extension of KAOS to reasoning about confidentiality requirements [30], Secure Tropos [48, 108, 110], and Secure i* [98].

KAOS: van Lamsweede [143] proposed an approach to modelling, specifying, and analysing security requirements. The approach extends an earlier framework on eliciting goals and identifying potential obstacles to satisfying goals [144] to security engineering. This is achieved by addressing malicious obstacles which could be setup by attackers to sabotage the satisfaction of security goals.

The security obstacles are called anti-goals and are similar to the idea misuse cases [134]. The anti-goals are then refined and elaborated through attack trees until leaf nodes which represent software vulnerabilities are identified. In a comparative study reported in Opdahl and Sindre [113], attack trees were found to be more effective at identifying security threats than misuses cases. In order to protect a system for the identified potential threats that could take advantage of the vulnerabilities, new security requirements are then elicited whose implementation is a countermeasure to potential threats.

Building on KAOS De Landtsheer and van Lamsweerde [86] proposed an approach to formally specifying and reasoning about confidentiality requirements in the early stages of software development. Confidentiality is one of key security requirements for information systems and it entails ensuring that information is accessible only to those authorised to access [67]. De Landtsheer and van Lamsweerde's approach makes it possible for requirements analysts to check requirements models for violation of confidentiality properties. In the case that violation of confidentiality properties is detected, a temporal sequences of state transitions is given that explains how confidential information could be disclosed.

The approach requires an analyst to specify all necessary confidentiality requirements. However, it is not always feasible to exhaustively eliciting all confidentiality requirements. This could be tackled by incorporating the approach into some form of analysis-revision cycle, such as Garcez *et al*. [29], that would allow for the continuous evolution of confidentiality requirements. Although the approach offers promising results for elicitation and analysis of confidentiality requirements its focus on confidentiality of state variables is limiting. Confidentiality of agent behaviour and what information can be inferred from such behaviour is another important aspect that the approach needs to take into account as a complement to the confidentiality of state variables. In addressing this limitation that approach may need to incorporate approaches to reasoning about the behaviour of contexts [52, 64] and how that behaviour contributes to satisfying or violating confidentiality requirements.

De Landtsheer and van Lamsweerde view of confidentiality is limited. Other works on elicitation and analysis of confidentiality requirements have proposed detailed classifications of confidentiality properties, albeit, in specific domains. For partitions of kernels in embedded software systems, Heitmeyer *et al.* [59] proposed at least four confidentiality properties: No-Exfiltration, No-Infiltration, Temporal Separation, and Separation of Control. The *No-Exfiltration* property states that data processing in a partition cannot influence data store outside that partition while the *No-Infiltration* property states that data processed in one partition is not influenced by data outside that partition.

The *Temporal Separation* property ensures that no confidential data stored in a partition in one configuration of the partition can remain in any memory area of that same partition in a later configuration. This property is similar to De Landtsheer and van Lamsweerde [86] view of confidentiality which states that an agent knows some data if it is stored in its memory. The *Separation of Control* property states that data processing between partitions is exclusive, that is, when data processing is in progress in one partition, there should no data processing in other partitions. For shared systems Jacob [67] proposed similar confidentiality properties, namely: non-interference, restriction of information flow, lack of strong flow, separability, and non-deducibility.

Another issue worth exploring is whether the conceptual basis of De Landtsheer and van Lamsweerde's approach can be extended for reasoning about other types of security requirements such as authenticity, availability, and non-repudiation at requirements engineering time.

Secure Tropos: Tropos [48, 108, 110] is a software development methodology, tailored from the i* modelling framework, which describes the system and its organisational environment. Models in Tropos are based on three basic concepts: actor, intention (goal, plan, resource), and dependency. An *actor* is an entity that has intentions within a system or organisation to be served by the system. A *goal* represents and actor's aim or purpose and a *plan* represents a means by which a goal maybe satisfied. A *resource* is a physical or informational entity that may be as used by the actions in a plan to satisfy a goal. When an actor depends on another to satisfy its goal, the two actors are said to have a *dependency* relationship. The Tropos software development process identifies four phases that to be followed in the development of software systems, namely: early requirements, late requirements, architectural design, and detailed design. Each successive phase refines the high-level description from the previous phase to a lower level towards implementation. The concepts and software development phases of Tropos have two limitations: (1) they fail to adequately capture security requirements and (2) they fail to provide concepts and processes for modelling trust relationships between actors.

In addressing the above limitations, Secure Tropos extends Tropos with the ability to model security concerns throughout the identified software development phases. This is achieved by explicit modelling of security constraints, secure entities trust of permission, and delegation of permission. A *security constraint* is a restriction related to maintaining security properties such as confidentiality and integrity. Security entities represent goals, tasks, or resources in an application. Secure *trust of permission* represents that a trust relation between two actors involves the introduction of a security constraint that must be satisfied for the trust relation to be considered valid. Similarly, secure *delegation of permission* involves the introduction of a security constraint that must be satisfied either for the delegating actor, or delegatee, or both for the delegation to be valid.

Although Secure Tropos provides a systematic methodology for eliciting and analysing security requirements, it does not provide means for propagating changes between the different models. For example, if there is a change in a trust of permission model there is no systematic way of relaying such

changes to a delegation of permission model, security constraint model, or security entities model. A clear interaction relationship between the models would provide a systematic way of propagating changes between the different models and hence support maintaining security properties as requirements evolve.

Secure i*: Liu *et al.* [98] proposed an approach to analysing security and privacy requirements based on the agent-oriented requirements modelling language i*. The main assumption in this approach is that security issues in a software application are a manifestation the relationships between the system's actors. Based on this assumption, the approach elicits and analyses security requirements through studying the relationships between system stakeholders, potential attackers, and agents acting either on behalf of attackers or stakeholders.

The approach consists of techniques for analysing attackers, dependency vulnerability, countermeasures, and access control. *Attacker analysis* identifies potential attackers and their possible malicious intents while *dependency vulnerability analysis* elicits vulnerabilities of an application based on relationship among stakeholders. *Countermeasure analysis* identifies measures that may be taken to prevent potential attacks and vulnerabilities from being realised. Finally, *access control analysis* establishes links between security requirements models and security implementation models by using i* models to refine proposed solutions and generate system designs that meet the security requirements. Trujillo *et al.* [139] proposed an approach for eliciting and developing security requirements for secure data warehouses which adapts the i* framework so that it can be used under model driven architecture and process modelling approaches.

The Secure i* approach is composed of useful abstractions of basic concepts of security such as identification of malicious actors and developing appropriate countermeasures. However, there are some limitations inherent in this approach. Firstly, while it is plausible that security issues manifested in a software application are a reflection of the dynamic behaviour of actors and their relationships in the social setting of an application, there is no guarantee that all potential classes of attackers can be identified. The implication of this is that the countermeasures taken are likely to be incomplete.

Secondly, the identification of vulnerabilities through dependencies between stakeholders is not foolproof because not all system vulnerabilities are due to relationships between stakeholders. Some security vulnerabilities may result from the addition of new features which may comprise the integrity of an application by violating security requirements maintained by current features. As an illustration consider the following example of conflicts between features in an automobile:

> Consider a car which has an alarm system (*security feature*) and a crash protection system with air bags (*safety feature*). The alarm system enforces security of the car occupants and their valuables. When activated it ensures that the doors and windows are locked; and monitors the state of the doors; and reports any burglary activity by activating the siren. Meanwhile, the safety feature ensures that in case of a crash, there is minimal loss of life. It achieves this by unlocking all doors in the event that an impact occurs on the front bumper.
>
> Let us consider a scenario where these features could interact. Assume the car is stationery at a traffic intersection with all doors locked by the *Security* feature. If a thief hits the front bumper with a big hammer, the *Safety* feature will unlock the doors allowing the thief to gain entry into the car.

This feature interaction may not be obvious to detect until a scenario such as the one above occurs and illustrates a situation in which an attacker takes advantage of the vulnerability resulting from evolution of an application by adding new functionality. A possible implication of this example could be that vulnerability analysis should not only consider relationships among actors but should also scenario-based misuse cases that may violate security requirements.

4.2 Model-Based Approaches

Model-Based approaches are based on the notion that models help requirements analysts in understanding complex software problems and identifying potential solutions through abstraction [34]. For example, models have been successfully for abstracting source code into class diagrams in reverse engineering. Such abstractions make it easier to understand the behaviour of a software system and how it might be improved. In this section we review two model-based approaches (UMLsec and

SecureUML) in security engineering. While there may be other model-based approaches aimed at addressing security concerns in the literature, our focus on these two is purely on a representational basis.

UMLsec [70]: This approach is an extension of UML which allows an application developer to embed security-related functionality into a system design and perform security analysis on a model of the system the system to verify that it satisfies particular security requirements. Security requirements are expressed as constraints on the behaviour of the system and the system design may be specified either in a UML specification or annotated in source code. Automated theorem proving or model checking is used to establish whether security requirements hold in the design. If the design violates security requirements, a Prolog-based tool is used to generate a scenario (in the form of attack sequences) of how security requirements may be violated by the design and countermeasure are taken to remove the vulnerability. UMLsec has been validated through its application to systems in mobile communications [73], automotive [13], and banking [69].

In essence, UMLsec assumes that requirements have already been elicited and there exists some system design to satisfy them. Its objective is to establish whether the system design satisfies security properties. The design is then progressively refined to ensure that it satisfies security requirements. However, reasoning about security requirements in model-based approaches relies on accuracy of system design models. The assumption that design models accurately capture system behaviour may not always hold and incompleteness in a model may leave vulnerabilities that are outside the scope of the model undetected.

For example, if a UML model used in the verification, obtained by reverse engineering source code, is not a true representation of the behaviour of the source code then it is inevitable that the results of verifying the design may not be accurate. In the case that verification suggests that the design does not satisfy security requirements it may not be possible to tell whether this is due to inaccurate translation to the design or the original source code. Of course, this may be seen as a concern that is outside the scope of model-based security engineering, but there is a need to ensure that the verified models are an accurate representation of reality. Some approaches have been proposed to verifying UMLsec models with system behaviour. These include using run-time verification [75], static formal verification [71], model-based testing [74], and security policy enforcement [61].

SecureUML: Lodderstedt *et al*. [99] present a modelling language, based on UML, called SecureUML. SecureUML is focused on modelling access control policies and how these (policies) can be integrated into a model-driven software development process. It uses role-based access control (RBAC) as a metamodel for specifying and enforcing security. Additionally, the language provides support for specifying authorisation constraints. The combination of the graphical capability of UML, access control properties of RBAC, and authorisation constraints makes it possible to base access decision on dynamically changing data such as time.

Similar to UMLsec, SecureUML focuses on the design phase of software development. For consistency, it is important to consider security using the same concepts and notations during the whole development process [108]. Unlike most approaches to security requirements analysis SecureUML does not offer any analysis of scenarios eliciting potential attacks. Its focus on authorisation constraints is insufficient as it does take into account potential vulnerabilities that could violate the constraints. Moreover, the approach does not provide for a systemic way of verifying the validity of the constraints.

4.3 Problem-Oriented Approaches

Problem oriented approaches [54, 55, 65], bring informal and formal aspects of software development together in a single theoretical framework for software engineering design – presenting development as the representation and step-wise transformation of software problems. This theoretical framework allows for: (1) the identification and clarification of system requirements; (2) the understanding and structuring of the problem world; (3) the structuring and specification of a machine that can ensure satisfaction of the requirements in the problem world; and (4) the construction of adequacy arguments, to convince both developers and other stakeholders that the system will provide what is needed. In this section we review three problem-oriented approaches, namely: security requirements and trust assumptions [52, 53], abuse frames [95, 96], and misuse cases [2, 3].

Security Requirements and Trust Assumptions: Haley *et al.* [52] proposed an approach to eliciting, specifying and analysing security requirements, which combines concepts from requirements engineering and securing engineering. From a requirements engineering perspective the approach uses the concept of functional goals which can be refined into functional requirements with relevant constraints. From a security engineering perspective, it takes the idea that security is about protecting assets from harm assets.

The approach consists of four components. The first component provides systematic statements of roles and relationships of security goals, security requirements and their relationships with other system requirements. The second describes threats and their potential interaction with the system. The third component is a precise definition of security requirements based on the description of potential threats. Finally, the fourth is a set of arguments which assists with validating whether the elicited security requirements can be enforced, given the context of the system. The construction of satisfaction arguments involves describing the system and its context in a problem diagram using Jackson's problem frames notation [64].

Abuse Frames: Lin et al. [95, 96] proposed abuse frames, an approach to analysing security problems in order to determine security vulnerabilities. This approach is also based on Jackson's problem frames approach to structuring and analysing software development problems [64]. While problem frames are aimed at analysing the requirements to be satisfied, in contrast, abuse frames are based on the notion of an anti-requirement. An anti-requirement is the requirement of a malicious user that can subvert an existing requirement (similar to the concept of an anti-goal [143]).

Abuse frames represent the notion of a security threat imposed by malicious users and a means for bounding the scope of security problems in order to analyse security threats and derive security requirements. Binding the scope of a security problem makes it possible to describe it more explicitly and precisely. Such explicit and precise descriptions facilitate the identification and analysis of threats, which in turn drive the elicitation and elaboration of security requirements.

Misuse Cases: Use cases document functional requirements of a system by exploring the scenarios in which the system may be used [68]. Their focus on what the system should do limits their capability in documenting security requirements, which often concentrate on what the system should not do [133]. Similar to abuse frames, misuse cases are a negative form of use cases and thus are use cases from the point of view of an actor hostile to the system [2, 3]. They are used for documenting and analysing scenarios in which a system may be attacked. Once the attack scenarios are identified, countermeasures are then taken to remove the possibility of a successful attack.

Although misuses cases are not entirely problem-oriented as they represent aspect of both problems and solutions, they have become popular as a means of representing security concerns in the early stages of software development. However, they are limited by the fact that they are based only on scenarios. Completeness of requirements analysed through scenarios is not guaranteed as other scenarios by which the security of a system could be exploited may be left out.

4.4 Process-Oriented Approaches

Process-oriented approaches focus on the steps for analysing security requirements. The steps may involve risk analysis for identifying security vulnerabilities and exploration of countermeasures for addressing identified weaknesses. In this section we review two process-oriented approaches: the SQUARE methodology and an aspect-oriented approach.

SQUARE [103]**:** The Security Quality Requirement Engineering (SQUARE) method is used for eliciting, analysing, categorising, prioritising, and documenting security requirements for software systems. The motivation of this method is to enable requirements analysts to identify security requirements as part of the requirements engineering process rather than as an after thought. The major stages of the method involve identifying security goals, performing risk analysis to identify potential threat to security goals, and eliciting security requirements which should be satisfied in order for security goals to be met.

The definition of security requirements in the SQUARE methodology considers requirements as being at the system or software level. This definition does not consider the properties and behaviour of the

context in which an application operates. According to Jackson [64], a more concise definition of requirements should consider their context of operation as satisfaction of a requirement is expressed in terms of the state changes in the context. The steps provide by the methodology are "waterfall model" in nature, and this does not make a provision for iterations to revise security requirements and support the evolution of a system. Nevertheless, the methodology provides concrete systematic steps for eliciting and analysing security requirement risk-based approach.

Aspect-Orientation: In Georg *et al.* [40] an aspect-oriented approach to designing secure applications is proposed. The main idea of the approach is modelling security mechanisms and attack models as aspects and consists of four steps: risk analysis, misuse model generation, composed system misuse model generation, and alternative solution analysis. *Risk analysis* involves analysing a system to identify potential threats to assets and the threats are modelled as attack aspects. *Misuse model generation* composes attack aspect with the base model of the application to create potential misuse models. Composed system misuse model generation analyses the misuse models to evaluate the impact of an attack. If the result of the evaluation is such that the impact of a potential attack is severe and cannot be tolerated, then countermeasures are identified through *alternative solution analysis*.

There are several benefits claimed for this approach. First, it allows designers to analyse and understand security mechanism and attack model in separately and in a modular form, thus making it easier to maintain the models. Second, using aspect composition and analysis techniques, designers can determine the effect of security mechanisms and attacks on other system functionality. Finally, determining the effects of new security mechanisms and potential attacks becomes a question of composing them with the existing application re-doing the aspect analysis. A similar approach is described in Xu et al. [150].

4.5 Summary of Approaches to Security Engineering

We have reviewed the state of the art of approaches to security requirements engineering. The approaches have been classified into goal-based, model-based, aspect-oriented, problem-based, and agent-oriented. Table 4 presents a comparative summary of these approaches and brief summaries of their main characteristics.

In Table 4, the *conceptual classification* column represents categories of security requirements engineering approaches based on the conceptual or primary characteristics for each security engineering approach. In this paper we have identified four categories, namely: goal-based, model-based, problem-oriented, and process-based approaches. Each conceptual class has instances or example approaches that fall in that category. These instances are listed in the *security approach* column.

Table 4. Comparative Summary of Security Requirements Engineering

Conceptual Classification	Security Approach	Core Representation	Security Specific Representation	Security Analysis	
				Vulnerability Identification Technique	Counter measure
Goal-Based	KAOS[143]	Goals	Anti-goals	Identification of Attacker Goals.	Elicitation of security goals to counter anti-goals.
	De Landtsheer and van Lamsweed [30]	Goals	Unauthorised Agent, Attacker Knowledge, and Confidentiality Requirements Patterns.	Requirements models are checked for violation of confidentiality properties.	Elicitation of security requirements to minimise the violation of confidentiality properties.
	Secure Tropos	TROPOS (Task, Actor, Resource, Goal, Soft Goal, Dependency)	Security constraint, secure entities, secure trust, and secure delegation modelling	Identification of malicious actor's goals and plans, and analysis of each actor's security constraints.	Revision of actor diagram to ensure that secure entities are protected from malicious actors
	Secure i*	Actor, goal, soft goal, task, resource, and belief.	Potential attacker, dependency vulnerability, and trust modelling,	*Attacker analysis* (identification of potential system abusers and their intents) and *Dependency Vulnerability Analysis* (identification of vulnerable points in actor dependency network)	i* models are used to refine proposed solution and generate system designs.

UML Model-Based Aspect-Oriented	UMLsec [72]	UML	Stereotypes	Theorem Proving, Model Checking.	System designs are revised to remove vulnerability.
	SecureUML [99]	UML and RBAC	Security Constraints	Modelling access control policies.	Identification of authorisation constraints
Problem-Based	Haley et al. [52]	Problem Frames	Trust assumption, assets domain, threat domain, warrants, grounds, satisfaction argument, and rebuttal.	Elicitation of *security goals* (based on possible harm to assets), refinement of security goals to *security requirements*, and construction of *satisfaction arguments*.	Removing rebuttal by adding functionality to permit the addition of new grounds or warrants to mitigate the conditions that permit a rebuttal.
	Abuse Frames [95]	Problem Frames	Anti-Requirements, Malicious User, Asset Under Attack, Security Requirement, Protected Domain	Problem frames are used as a means of security threat analysis and identifying anti-requirements. Security threats are then expressed as abuse frames.	Identification of abuse frame concerns which need to be addressed for an attack to succeed. Security requirement for counteracting threats are expression a problem frame.
	Misuse Cases [3]	Use Case	Misuse Case		
Process-Oriented	SQUARE [103]	Goals and Risk	Misuse Cases, Attack Scenarios, Attack Trees.	Risk assessment to misuse case, attack scenarios, attack scenarios, attack trees.	Elicitation of security requirements from potential risks.
	Georg *et al*. [40]	Aspects and Risk	Secure aspect	Misuse Model Generation	Alternative Solution Analysis

The *core representation* column shows the basic notation that each security engineering approach uses for expressing its basic concepts. For example the basic construct in goal-oriented approaches is the goal notation. Security requirements have special characteristics which often make it necessary to extend generic core representations with security specific notations that are tailored for capturing and representing security concerns.

For each approach security-related concepts are presented in the *security representation* column. In general approaches to security requirements engineering involve two main phases in their processes, namely (1) identification of vulnerabilities of a system to security threats, their probability of occurrence, and impact and (2) designing mitigation strategies to remove the possibility of threats causing harm to assets. The *security analysis* column documents methods and techniques by which each approach identifies system vulnerabilities and identify or design counter measures to potential threats.

The next section evaluates the extent to which approaches to security requirements engineering support software evolution.

5. Support for Software Evolution in Security Requirements Engineering Approaches

Security engineering and software evolution, although often conflicting, are intertwined in the sense that a change in one may affect the other. For example a violation of security goals may result in new security requirements as countermeasures which in turn lead to an evolution of system functionality. Likewise, the inevitable evolution of a system may lead to the addition of new functionality which violates security properties.

In this section we make a comparative evaluation of the main characteristics of the security requirements engineering approaches we reviewed in section 4. Our evaluation is based on a comparison criterion that examines support for software evolution in security engineering approaches. In the introduction section, we suggested that one way for security approaches to address concerns of evolution in long-lived systems is to integrate software evolution management approaches in security engineering. How can this be achieved?

In order to address the question above, we need to know what exactly is missing in security requirements engineering approaches that can be considered sufficient in order to address software evolution concerns. To systematically elicit the limitations of security engineering approaches we did an analytical comparative evaluation and we started by formulating a possible comparison criteria. The purpose of the criterion was to evaluate to what extent do current approaches to security requirements engineering support software evolution. In formulating the criterion we had to ask a more fundamental question: *what would it mean for a security engineering approach to support software evolution?*

Our discussion in section 2 centred on the notion that the core element of software evolution is change. Thus, supporting the evolution of a long-lived system mainly concerns tailoring its architecture and life cycle in such a way that it makes it easier to accommodate and manage changing requirements. Our evaluation criterion consists of four dimension of software evolution which we consider important for security engineering approaches. In section 5.1 we present these criteria and discuss the evaluation results in section 5.2.

5.1 Evaluation Criteria

We identified four dimensions for evaluating support of software evolution in security requirements engineering approaches. These are modularity, component architecture, change propagation, and change impact analysis. We briefly explain these below.

Modularisation: This is one of the most fundamental software engineering design principle. The value of software design modularity mainly lies in the ability to accommodate potential changes. Modularization techniques, such as object-oriented design patterns, provide one way to make some part of a system change independently of all other parts. Modularity enforces separation of concerns and makes it possible to develop software components independently and assembly them later. Constructs such as features, classes, objects, components, and aspects are all means to modularisation. Encapsulation is a key factor in modularisation as it determines the ability of a system design to contain changes within a single module. For software evolution modularisation is important because, potentially, it makes it easier to change the functionality of a software system by making it possible to add or remove components.

Component Architectures: Modularity alone is insufficient in supporting software evolution in long-lived systems. It is necessary to have an infrastructure where the software modules can be added and removed with ease [117]. Component architectures provide such infrastructure by offering mechanisms for component interoperability and integration which make it possible to extend systems with third party components and hence provide support for evolution.

Change Propagation: The feature driven development paradigm [116] organises the functionality of a software system in terms of features which may have dependencies between each other. One feature A is dependent on some other feature B if B provides some services that A requires for its correct operation. This dependency implies that changes in feature B may affect feature A. For example a method in a class in B is changed and that method is called by A, the method call in A has to change accordingly – otherwise we have an inconsistent dependency. In order to correct such inconsistencies, further changes have to be made until consistency is restored. A change propagation process keeps track of these changes and help in guaranteeing that a change is correctly propagated and that no inconsistent dependency is left unresolved.

Change Impact Analysis: This is similar concept to change propagation. While the change propagation is concerned with recording assessing the ripple effect of changes, the objective of impact analysis is to determine what would be affected by a change to a particular artefact [15, 58]. This involves identifying the artefact to be changed and how other artefacts that depend on it (its dependent relationship). Identifying, dependent relationship is a recursive process as artefacts that depend on the selected artefact may also have their own dependents, and so on. The process of dependency relationships analysis continues until all dependencies are identified, starting with the selected artefact and finishing with artefacts where nothing else depends on it.

5.2 Evaluation results

Table 5 presents a comparative evaluation of the security requirements engineering approaches discussed in section 4 using the evaluation criterion above. The evaluation of each approach is based on analysing the characteristics of the core representation, security specific representation, vulnerability identification technique, and countermeasure techniques to accommodating change. The aim is establish the extent to which current approaches to security engineering support software evolution and whether some of their aspects hinder support for change.

We evaluate each approach by assigning an integer value in the range 0 to 3. At the lower end, the value 0 implies that an approach offers little or no support for a particular aspect of software evolution. On the higher end of the scale, the value 3 implies that an approach fully supports the given aspect of evolution.

Table 5. Evaluation of Support for Software Evolution in Security Requirements Engineering Approaches

Conceptual Classification	Security Approach	Security Evolution Support			
		Modularity	Component Architectures	Change Propagation	Change Impact Analysis
Goal-Based	KAOS[143]	2: The decomposition of a system into goals supports modularity.	0: There is no explicit support for component architectures.	3: A goal model shows the relationship between goals and hence their dependencies.	1: There is no explicit support for change impact analysis as the focuss is one identifying threats to existing goals (rather the effect of adding new goals)
	De Landtsheer and van Lamsweed [30]	1: Goals are used as a construct for modularity	0: There is no explicit consideration for component infrastructures.	3: Dependencies between goals are modelled in a goal model.	1: There is no explicit support. Focussed on identifying violation of confidentiality by existing goals.
	Secure Tropos	1: Although agents are used for identifying attackers, goals are the main unit of modularity.	0: Component infrastructures are not explicitly supported.	1: It is possible to analyse dependency relationships between agents.	1: There is no explicit support for analysing the impact of adding new goals.
	Secure i*	1: same as for SecureTropos.	0: There is no explicit support for component architectures.	3: Achieved by modelling dependencies between stakeholders.	1: Although there is support for analysing the security impact of existing goals, there is no explicit support on how the impact of adding new goals is analysed.
Model-Based	UMLsec [72]	2: Support is implicit as it is dependent on the OO nature of UML design models.	2: This is implicit in UML, although the approach does not prescribe architectures. It verifies existing designs.	2: Support is implicit as it is depended on the language used for the modelling language.	3: Model-Checking and Theorem proving techniques are used to verify the impact of change.
	SecureUML [99]	2: There is implicit support from the component nature of UML.	2: This provided by UML.	2: Same as for UMLsec	1: There is no explicit support, although new functionality can be verified against authorisation constraints.
Problem-Oriented	Haley et al. [52]	2: Modules are represented as problem descriptions.	1: Focus is on eliciting security requirement rather how problem can be composed.	1: There is no explicit modelling for dependencies between functions	3: Argument satisfaction is used as a way of verifying that a specification satisfies a requirement in a given context.
	Abuse Frames [96]	2: Modules are represented as problem descriptions.	1: There is no explicit support for this. Depends on the structure of the system analysed.	1: There is no explicit support for change propagation.	2: Although there is explicit support, change impact analysis can be achieved problem analysis when new security problems are identified.
	Misuse Cases [3]	2: Modules are use cases.	1: There is no explicit support for component architectures.	0: Focus is on identifying potential system abuses than interaction between functions	2: This is implicit in the approach as it possible to identify misuse cases for corresponding to use cases.
Process-Oriented	SQUARE [103]	0: There is no support for modularity. Focus is on risk analysis.	0: The approach is focussed on steps for risk analysis independent of the underlying structure of the systems analysed.	3: Risk analysis identifies dependencies, however, not necessary for change propagation.	3: Although, the steps in the approach are 'water model' like rather than iterative, the approach can be used for impact analysis.
	Georg *et al*. [40]	2: The aspect is the construct for modularity.	1: Aspect weaving techniques provide a way to compose aspects.	3: Aspects encapsulate cross-cutting concerns, hence show dependency between components.	1: Focus is on encapsulating security concerns in aspects. There is no explicit support for change impact analysis.

We noted in table 3.2 that none of the approaches to managing software evolution we have reviewed consider security concerns. It is worth noting, in Table 5, that some approaches to security requirements engineering approaches discussed seems to provide some limited support for software evolution. More comprehensive support is necessary as software evolution is a survival characteristic for long-lived systems.

As stated earlier, evolution often leads to violation of security requirements. Therefore there is need to investigate how software evolution could be a part of security requirements engineering and vice versa. In the next section we present a research agenda for security requirements for evolving systems.

6. Security Requirements Engineering for Evolving Systems: A Research Agenda

Based on our review of software evolution and security engineering, in this section we articulate open researches issues and present a research agenda in security requirements engineering for evolving systems. We frame the open research issues around challenges in both software evolution and security requirements engineering, and where possible, highlight some promising ideas on how the issues arising from the integration of evolution and security engineering may be addressed. Our discussion of the challenges is based on previous works Mens et al. [106] and Mouratidis and Giorgini [107]. While these works focussed on software evolution and security engineering, respectively, the theme of our discussion is how to maintain satisfaction of security requirements while supporting continuous evolution of software systems.

Understanding Change: As evolution in a software system is a manifestation of the changes in an organisation, there is a need to understand and capture evolution not only at the level of a software system but also at the organisational level. Current approaches to studying software evolution are focussed more evolution at the system-level rather instead of organisation level.

There are several benefits to understanding change at organisation level. The most important is that organisations often state their needs in terms of visions and goals. In contrast to low-level requirements visions and goals are more stable in the face of change. Due to such stability is it therefore important that long-lived systems evolution should be understood in terms visions and goals instead low-level requirements evolution. An understanding of change could also lead to a theory of software evolution which could explain whether systems evolve functionally in non-random and partly predictable ways. Such theory could also make it feasible to integrate change as part of the software lifecycle. Support for model evolution [106] is one of the key challenges in software evolution that could also benefit from a high-level understanding of change.

Brier *et al.*'s [19] approach to capturing, analysing, and understanding how software systems adapt to changing requirements in their organisational context. Although their approach is aimed at re-aligning software systems to business goals and processes; it can be seen as a step towards understanding change at organisation level. The approach includes a process of change analysis for evaluating improvements resulting from change in an organisation and a notation for reasoning about change.

Designing Change Tolerant Software Systems: Changing user needs induce new requirements and technological advances may require a change in the context of an application. Evolution of an application is inevitable and software systems often break due to changes resulting from evolution. There is need for an approach to designing software systems in such a way that they can tolerate change, that is, they are evolvable and their evolution does not lead to failure.

Promising approaches to designing change tolerant systems include: Ravichandar et al.'s [123] capabilities-based approach to designing change tolerant systems; Zave's [153] feature-based and component-centric architecture approach to evolving software systems; Zowghi [157] approach to modelling and reasoning about requirements evolution; and Garcez *et al.* [29] to evolving specifications. Another promising approach is described in Shin and Gomaa [132]. The approach models the evolution of non-secure applications into secure applications in terms of the software requirements model and software architecture model. Security requirements are captured separately from functional requirements and it is claimed that this separation makes possible to achieve the evolution from a non-secure application to a secure application with less impact on the application.

Non-Monotonicity of Software Evolution: Achieving systems that are secure and evolvable is a hard goal because software evolution and security are conflicting goals [111]. One of the key characteristics of software evolution is that in response to new requirements, new features may be added to legacy systems. This mandates composition of the existing feature set with new features. However, feature composition is non-monotonic [146] due to the feature interactions problem [78]. A system is said to be Non-monotonic if it does not guarantee that properties that held prior to addition of new functionality will continue to hold after the functionality has been added [56].

Since software evolution involves the composition of existing features with new features, and feature composition is non-monotonic, then software evolution is intrinsically a non-monotonic activity. Therefore, one of the important challenges for security engineering for evolving systems is how to balance between the inevitable need for supporting continuous software evolution and the goal of designing systems which ensure that security requirements that held initially (and need to continue holding) are not violated by the addition of new functionality. This challenge can be summarised as follows: can continuous software evolution co-exist with stringent security requirements and how can this be achieved through sound design principles, methods, languages, and tools? How can vulnerabilities resulting from the addition of new features be minimized?

Garcez et al [29] approach of analysis and change (as described in section 2.4) holds some promise as it makes it possible for systems to be evolved in such a manner that allows the satisfaction of desirable requirements to be checked at the end of an evolution cycle. At its present state, this approach allows for the violation of security properties and then evolving the specification to remove the violation. This is not a desirable characteristic especially in cases where the effects of the violation of a security requirement can not be reversed. An interesting challenge is how this approach (other similar approaches) could be modified such that evolutionary changes are only permitted only if the implication of any resulting violation to security requirements is minimal. This may involve taking into account the physical context of operation. This could be achieved by combining a analysis and revision approaches with problem-oriented approaches security requirements engineering (such as those proposed by Haley *et al*. [52] and Salifu *et al*. [130]), and incorporating promising results from secure software composition [9, 10, 37, 38, 101, 102].

Security for Evolving Context-Aware Software: Context-aware applications have to maintain satisfaction of requirements despite changes in their operating conditions [130]. Designing context-aware systems involves analysing possible variations in their context of operation and specifying behaviours in advance that would enable the system to maintain satisfaction of its requirements despite changes in context. Besides the repository of behaviours corresponding to different context, context-aware systems are also equipped with mechanisms for monitoring their context and switch between behaviours in response to contextual changes. Evolutionary changes in a context-ware application are often driven by the introduction of a new context of operation that had not been considered initially. This makes it necessary to specify new behaviours to enable the application to continue to operate in the new context and a specification of variables to be monitored in the new context.

Research in context-ware systems is relatively new. As a result current approaches to managing software evolution are focussed on systems that do not need to change their behaviour with changes in context. We envisage that the adaptive and dynamic nature of context-ware applications brings to fore additional concerns and challenges for both software evolution and security engineering. In software evolution one of the important research issues is whether the approaches proposed for managing evolution in none context-ware systems can be applied to context aware systems. There are at least two perspectives from which software evolution in an adaptive environment can studied. One concern involves evolution of system behaviour with changing context. The other relates to evolution in terms of new behaviour introduced to an application due to new context that was not considered initially. It is worth investigating the interaction between these perspectives of evolution and the security concerns they may raise.

An even harder challenge of security and evolution in context-aware systems is online software evolution [148], which is a kind of software evolution that updates running programs without interruption of their execution. Evolution for such systems is dynamic and often has to be completed in relatively short time limits. This timing constraint raises at least two concerns. (1) How can the correctness of evolved software be verified? Current approaches to verification are based on model checking and theorem proofing [21, 33, 47, 93]. Both of these verification techniques are resource

intensive operations and often take long to complete. (2) If the event that the online evolution fails, can the evolution be rolled back? What are the implications of such roll back on security properties?

7. Conclusion

Software systems evolve in response to changes in their operating environment and requirements. Such evolution often violates security requirements. We have reviewed the state-of-the-art in security engineering and concluded that current approaches to security engineering do not address the problem of preserving security properties that may be violated as a result of software evolution.

This paper suggested that one approach to addressing this problem of preserving security properties is to integrate approaches to managing software evolution in security engineering. We termed this as security requirements engineering for evolving systems. We have identified and discussed open research issues and challenges that may need to be addressed in order to achieve the goal of security engineering for evolving software systems. In some cases we have discussed promising research directions on how the identified open issues could be addressed.

References

1. Ahmad, A., H. Basson, and M. Bouneffa. *Software evolution control towards a better identification of change impact propagation*. in *4th International Conference on Emerging Technologies*. 2008.
2. Alexander, I. *Initial industrial experience of misuse cases in trade-off analysis*. in *Proceedings of IEEE Joint International Conference on Requirements Engineering*. 2002.
3. Alexander, I., *Misuse cases: use cases with hostile intent.* IEEE Software, 2003. **20**(1): p. 58-66.
4. Anton, A.I. and C. Potts, *Functional Paleontology: The Evolution of User-Visible System Services.* IEEE Transactions on Software Engineering, 2003. **29**(2): p. 151-166.
5. Antón, A.I. and C. Potts. *Functional Paleontology: System Evolution as the User Sees It*. in *23rd International Conference on Software Engineering (ICSE'01)*. 2001.
6. Antón, A.I. and C. Potts. *Requirements Engineering in the Long-Term: Fifty Years of Telephony Feature Evolution*. in *International Workshop on Feedback and Evolution in Software and Business Processes (FEAST 2000)*. 2000. London, UK, 10-12 July 2000.
7. Antonellis, P., D. Antoniou, Y. Kanellopoulos, C. Makris, E. Theodoridis, C. Tjortjis, and N. Tsirakis, *Clustering for Monitoring Software Systems Maintainability Evolution.* Electronic Notes in Theoretical Computer Science, 2009. **233**: p. 43-57.
8. Barry, E.J., C.F. Kemerer, and S.A. Slaughter, *How software process automation affects software evolution: a longitudinal empirical analysis.* Journal of Software Maintenance and Evolution: Research and Practice, 2007. **19**(1): p. 1-31.
9. Bartoletti, M., P. Degano, and G.L. Ferrari. *Enforcing secure service composition*. in *18th IEEE Workshop Computer Security Foundations*. 2005.
10. Bartoletti, M., P. Degano, G.L. Ferrari, and R. Zunino, *Semantics-Based Design for Secure Web Services.* IEEE Transactions on Software Engineering, 2008. **34**(1): p. 33-49.
11. Becker, S.M., T. Haase, and B. Westfechtel, *Model-based a-posteriori integration of engineering tools for incremental development processes.* Software and Systems Modeling, 2005. **4**(2): p. 123-140.
12. Bergmann, G., A. Okros, I. Rath, D. Varro, and G. Varro, *Incremental pattern matching in the viatra model transformation system*, in *Proceedings of the third international workshop on Graph and model transformations*. 2008, ACM: Leipzig, Germany. p. 25-32.
13. Best, B., J. Jurjens, and B. Nuseibeh. *Model-Based Security Engineering of Distributed Information Systems Using UMLsec*. in *29th International Conference on Software Engineering*. 2007.
14. Bezivin, J., *On the Unification Power of Models.* Software and Systems Modeling, 2005. **4**(2): p. 171-188.
15. Bohner, S.A. *Software change impacts-an evolving perspective*. in *Proceedings of International Conference on Software Maintenance*. 2002.
16. Bond, G.W., E. Cheung, K.H. Purdy, P. Zave, and C. Ramming, *An Open Architecture for Next-Generation Telecommunication Services.* ACM Transactions on Internet Technology (TOIT), 2004. **4**(1): p. 83-123.
17. Bradbury, J.S., J.R. Cordy, J. Dingel, and M. Wermelinger, *A survey of self-management in dynamic software architecture specifications*, in *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*. 2004, ACM Press: Newport Beach, California. p. 28-33.
18. Breaux, T.D. and A.I. Anton, *Analyzing Regulatory Rules for Privacy and Security Requirements.* IEEE Transactions on Software Engineering, 2008. **34**(1): p. 5-20.

19. Brier, J., L. Rapanotti, and J.G. Hall. *Problem-based analysis of organisational change: a real-world example*. in *Proceedings of the 2006 international workshop on Advances and applications of problem frames*. 2006. Shanghai, China: ACM.
20. Calder, M., M. Kolberg, E. Magill, and S. Reiff-Marganiec, *Feature interaction: A critical review and considered forecast.* Computer Networks, 2003. **41**(1): p. 115-141.
21. Calder, M. and A. Miller, *Feature interaction detection by pairwise analysis of LTL properties: a case study.* Formal Methods in System Design, 2006. **28**(3): p. 213-261.
22. Chakraborty, D., F. Perich, A. Joshi, T.W. Finin, and Y. Yesha. *A Reactive Service Composition Architecture for Pervasive Computing Environments*. in *Proceedings of the IFIP TC6/WG6.8 Working Conference on Personal Wireless Communications*. 2002: Kluwer, B.V. Deventer, The Netherlands, The Netherlands.
23. Chung, L. and N. Subramanian, *Architecture-based semantic evolution of embedded remotely controlled systems.* Journal of Software Maintenance and Evolution: Research and Practice, 2003. **15**(3): p. 145-190.
24. Cook, S., R. Harrison, M.M. Lehman, and P. Wernick, *Evolution in software systems: foundations of the SPE classification scheme.* Journal of Software Maintenance and Evolution: Research and Practice, 2006. **18**(1): p. 1-35.
25. Cook, S., R. Harrison, M.M. Lehman, and P. Wernick, *Evolution in software systems: foundations of the SPE classification scheme.* Journal of Software Maintenance and Evolution: Research and Practice, 2005. **18**(1): p. 1-35.
26. Crnkovic, I. *Component-based software engineering - new challenges in software development*. in *Information Technology Interfaces, 2003. ITI 2003. Proceedings of the 25th International Conference on*. 2003.
27. Czarnecki, K. and S. Helsen, *Feature-based survey of model transformation approaches.* IBM Systems Journal, 2006. **45**(3): p. 621-645.
28. da Silva, B.C., E. Figueiredo, A. Garcia, and D. Nunes, *Refactoring of Crosscutting Concerns with Metaphor-Based Heuristics.* Electronic Notes in Theoretical Computer Science, 2009. **233**: p. 105-125.
29. d'Avila Garcez, A.S., A. Russo, B. Nuseibeh, and J. Kramer, *Combining abductive reasoning and inductive learning to evolve requirements specifications.* IEE Proceedings Software, 2003. **150**(1): p. 25-38.
30. de Landtsheer, R. and A. van Lamsweerde, *Reasoning about confidentiality at requirements engineering time*, in *Proceedings of the 10th European software engineering conference*. 2005, ACM: Lisbon, Portugal. p. 41-49.
31. Del Rosso, C., *Continuous evolution through software architecture evaluation: a case study.* Journal of Software Maintenance and Evolution: Research and Practice, 2006. **18**(5): p. 351-383.
32. Fabbrini, F., M. Fusani, S. Gnesi, and G. Lami. *Controlling Requirements Evolution: a Formal Concept Analysis-Based Approach*. in *International Conference on Software Engineering Advances*. 2007.
33. Felty, A.P. and K.S. Namjoshi, *Feature specification and automated conflict detection.* ACM Transactions on Software Engineering and Methodology (TOSEM), 2003. **12**(1): p. 3 - 27.
34. Fernández-Medina, E., J. Jurjens, J. Trujillo, and S. Jajodia, *Model-Driven Development for secure information systems.* Information and Software Technology, 2009. **51**(5): p. 809-814.
35. Fischer, M. and H. Gall, *Visualizing feature evolution of large-scale software based on problem and modification report data.* Journal of Software Maintenance and Evolution: Research and Practice, 2004. **16**(6): p. 385-403.
36. Fisler, K. and S. Krishnamurthi. *Modular verification of collaboration-based software designs*. in *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*. 2001. Vienna, Austria.
37. Focardi, R. and R. Gorrieri, *The Compositional Security Checker: a tool for the verification of information flow security properties.* IEEE Transactions on Software Engineering, 1997. **23**(9): p. 550-571.
38. Francesco, N.D. and G. Lettieri, *Checking security properties by model checking.* Software Testing, Verification and Reliability, 2003. **13**(3): p. 181-196.
39. Gall, H., M. Jazayeri, and C. Riva. *Visualizing Software Release Histories: The Use of Color and Third Dimension*. in *Proceedings of the IEEE International Conference on Software Maintenance*. 1999: IEEE Computer Society Washington, DC, USA.
40. Georg, G., I. Ray, K. Anastasakis, B. Bordbar, M. Toahchoodee, and S.H. Houmb, *An aspect-oriented methodology for designing secure applications.* Information and Software Technology, 2009. **51**(5): p. 846-864.
41. Georgas, J.C., A.v.d. Hoek, and R.N. Taylor, *Architectural runtime configuration management in support of dependable self-adaptive software*, in *Proceedings of the 2005 workshop on Architecting dependable systems*. 2005, ACM Press: St. Louis, Missouri. p. 1-6.

42. Gerber, A., M. Lawley, K. Raymond, J. Steel, and A. Wood, *Transformation: The Missing Link of MDA*, in *Proceedings of the 1st International Conference on Graph Transformation*. 2002, Springer-Verlag: Barcelona. p. 90-105.
43. Gerdes, J., *User Interface Migration of Microsoft Windows Applications.* Journal of Software Maintenance and Evolution: Research and Practice, 2009. **9999**(9999): p. n/a.
44. German, D.M., *Using software trails to reconstruct the evolution of software.* Journal of Software Maintenance and Evolution: Research and Practice, 2004. **16**(6): p. 367-384.
45. Ghose, A.K. *A formal basis for consistency, evolution and rationale management in requirements engineering*. in *11th IEEE International Conference on Tools with Artificial Intelligence*. 1999.
46. Ghose, A.K. *Formal tools for managing inconsistency and change in RE*. in *10th International Workshop on Software Specification and Design*. 2000.
47. Giannakopoulou, D. and J. Magee, *Fluent model checking for event-based systems*, in *Proceedings of the 9th European Software Engineering Conference*. 2003, ACM Press: Helsinki, Finland. p. 257-266.
48. Giorgini, P., F. Massacci, J. Mylopoulos, and N. Zannone. *Modeling security requirements through ownership, permission and delegation*. in *Proceedings of 13th IEEE International Conference on Requirements Engineering*. 2005. Paris, France.
49. Gîrba, T. and S. Ducasse, *Modeling history to analyze software evolution.* Journal of Software Maintenance and Evolution: Research and Practice, 2006. **18**(3): p. 207-236.
50. Glorie, M., A. Zaidman, A.v. Deursen, and L. Hofland, *Splitting a large software repository for easing future software evolution - an industrial experience report.* Journal of Software Maintenance and Evolution: Research and Practice, 2009. **21**(2): p. 113-141.
51. Greevy, O., S. Ducasse, and Tudor Gîrba, *Analyzing software evolution through feature views.* Journal of Software Maintenance and Evolution: Research and Practice, 2006. **18**(6): p. 425-456.
52. Haley, C.B., R. Laney, J.D. Moffett, and B. Nuseibeh, *Security Requirements Engineering: A Framework for Representation and Analysis.* IEEE Transactions on Software Engineering, 2008. **34**(1): p. 133-153.
53. Haley, C.B., R.C. Laney, J.D. Moffett, and B. Nuseibeh. *The effect of trust assumptions on the elaboration of security requirements*. in *12th IEEE International Requirements Engineering Conference*. 2004.
54. Hall, J.G., L. Rapanotti, and M. Jackson. *Problem Oriented Software Engineering: A design-theoretic framework for software engineering*. in *5th IEEE International Conference on Software Engineering and Formal Methods*. 2007.
55. Hall, J.G., L. Rapanotti, and M.A. Jackson, *Problem Oriented Software Engineering: Solving the Package Router Control Problem.* IEEE Transactions on Software Engineering, 2008. **34**(2): p. 226-241.
56. Hall, R.J., *Feature combination and interaction detection via foreground/background models.* Journal of Computer Networks, 2000. **32**(4): p. 449-469.
57. Hassan, A.E. and R.C. Holt. *Studying the evolution of software systems using evolutionary code extractors*. in *Proceedings of 7th International Workshop on Principles of Software Evolution*. 2004.
58. Hassine, J., J. Rilling, J. Hewitt, and R. Dssouli. *Change impact analysis for requirement evolution using use case maps*. in *8th International Workshop on Principles of Software Evolution*. 2005.
59. Heitmeyer, C.L., M.M. Archer, E.I. Leonard, and J.D. McLean, *Applying Formal Methods to a Certifiably Secure Software System.* IEEE Transactions on Software Engineering, 2008. **34**(1): p. 82-98.
60. His, I. and C. Potts, *Studying the Evolution and Enhancement of Software Features*, in *Proceedings of the International Conference on Software Maintenance*. 2000, IEEE Computer Society. p. 143.
61. Hohn, S. and J. Jurjens, *Rubacon: automated support for model-based compliance engineering*, in *Proceedings of the 30th international conference on Software engineering*. 2008, ACM: Leipzig, Germany. p. 875-878.
62. Hsi, I. and C. Potts. *Studying the Evolution and Enhancement of Software Features*. in *Proceedings of the 16th IEEE International Conference on Software Maintenance*. 2000. San Jose, California, USA.
63. Ibrahim, N., W.M.N. Wan Kadir, and S. Deris. *Comparative Evaluation of Change Propagation Approaches towards Resilient Software Evolution*. in *3rd International Conference on Software Engineering Advances*. 2008.
64. Jackson, M., *Problem frames : analysing and structuring software development problems*. ACM Press. 2001, Harlow: Addison-Wesley, 2001.
65. Jackson, M., *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices*. 1995, London, United Kingdom: Addison-Wesley. 228.
66. Jackson, M. and P. Zave, *Distributed Feature Composition: A Virtual Architecture for Telecommunications Services.* IEEE Transactions on Software Engineering, 1998. **24**(10): p. 831-847.
67. Jacob, J., *A uniform presentation of confidentiality properties.* IEEE Transactions on Software Engineering, 1991. **17**(11): p. 1186-1194.

68. Jacobson, I., *Object Oriented Software Engineering: A Use Case Driven Approach*. 1992: Addison-Wesley Professional. 552.
69. Jurjens, J. *Developing high-assurance secure systems with UML: a smartcard-based purchase protocol*. in *Proceedings of 8th IEEE International Symposium on High Assurance Systems Engineering*. 2004.
70. Jurjens, J., *Secure Systems Development with UML*. 2004, Heidelberg, German: Springer-Verlag. 316.
71. Jurjens, J. *Security Analysis of Crypto-based Java Programs using Automated Theorem Provers*. in *21st IEEE/ACM International Conference on Automated Software Engineering*. 2006.
72. Jurjens, J. *UMLsec: Extending UML for Secure Systems Development*. in *Proceedings of the 5th International Conference on The Unified Modeling Language*. 2002: Springer-Verlag.
73. Jurjens, J., J. Schreck, and P. Bartmann, *Model-based security analysis for mobile communications*, in *Proceedings of the 30th international conference on Software engineering*. 2008, ACM: Leipzig, Germany. p. 683-692.
74. Jurjens, J. and G. Wimmel. *Formally testing fail-safety of electronic purse protocols*. in *16th Annual International Conference on Automated Software Engineering*. 2001.
75. Jurjens, J., Y. Yu, and A. Bauer. *Tools for Traceable Security Verification*. in *Visions of Computer Science - BCS International Academic Conference*. 2008. Imperial College, London, UK: BCS.
76. Kagdi, H., M.L. Collard, and J.I. Maletic, *A survey and taxonomy of approaches for mining software repositories in the context of software evolution.* Journal of Software Maintenance and Evolution: Research and Practice, 2007. **19**(2): p. 77-131.
77. Kang, K.C., S. Kim, J. Lee, and K. Lee, *Feature-oriented engineering of PBX software for adaptability and reuseability.* Software: Practice and Experience, 1999. **29**(10): p. 875 - 896.
78. Keck, D.O. and P.J. Kuehn, *The Feature and Service Interaction Problem in Telecommunications Systems: A Survey.* IEEE Transactions on Software Engineering, 1998. **24**(10): p. 779-796.
79. Kemerer, C.F. and S. Slaughter, *An Empirical Approach to Studying Software Evolution.* IEEE Transactions on Software Engineering, 1999. **25**(4): p. 493 - 509.
80. Kitchenham, B.A., G.H. Travassos, A.v. Mayrhauser, F. Niessink, N.F. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, and H. Yang, *Towards an ontology of software maintenance.* Journal of Software Maintenance: Research and Practice, 1999. **11**(6): p. 365-389.
81. Kolberg, M., E.H. Magill, and M. Wilson, *Compatibility Issues between Services Supporting Networked Appliances.* IEEE Communications Magazine, 2003. **41**(11): p. 136-147.
82. Kosker, Y., B. Turhan, and A. Bener, *An expert system for determining candidate software classes for refactoring.* Expert Systems with Applications. **In Press, Corrected Proof**.
83. Kozlov, D., J. Koskinen, M. Sakkinen, and J. Markkula, *Assessing maintainability change over multiple software releases.* Journal of Software Maintenance and Evolution: Research and Practice, 2008. **20**(1): p. 31-58.
84. Lam, W. and M. Loomes. *Requirements evolution in the midst of environmental change: a managed approach*. in *2nd Euromicro Conference on Software Maintenance and Reengineering*. 1998.
85. LaMantia, M.J., Y. Cai, A. MacCormack, and J. Rusnak, *Analyzing the Evolution of Large-Scale Software Systems Using Design Structure Matrices and Design Rule Theory: Two Exploratory Cases*, in *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*. 2008, IEEE Computer Society. p. 83-92.
86. Landtsheer, R.D. and A.v. Lamsweerde, *Reasoning about confidentiality at requirements engineering time*, in *Proceedings of the 10th European software engineering conference*. 2005, ACM: Lisbon, Portugal. p. 41-49.
87. Lee, K., K.C. Kang, W. Chae, and B.W. Choi, *Feature-based approach to object-oriented engineering of applications for reuse.* Software: Practice and Experience, 2000. **30**(9): p. 1025-1046.
88. Lehman, M.M. *Laws of Software Evolution Revisited*. in *Proceedings of the 5th European Workshop on Software Process Technology*. 1996.
89. Lehman, M.M., G. Kahen, and J.F. Ramil, *Behavioural modelling of long-lived evolution processes - some issues and an example.* Journal of Software Maintenance and Evolution: Research and Practice, 2002. **14**(5): p. 335-351.
90. Lehman, M.M. and J.F. Ramil, *Evolution in software and related areas*, in *Proceedings of the 4th International Workshop on Principles of Software Evolution*. 2001, ACM: Vienna, Austria. p. 1-16.
91. Lehman, M.M. and J.F. Ramil, *Software evolution: background, theory, practice.* Information Processing Letters, 2003. **88**(1-2): p. 33-44.
92. Lemoine, M. and J. Foisseau. *Managing (requirements) evolutions of high assurance systems*. in *IEEE Joint International Conference on Requirements Engineering*. 2002.
93. Letier, E., J. Kramer, J. Magee, and S. Uchitel, *Fluent temporal logic for discrete-time event-based models.* SIGSOFT Softw. Eng. Notes, 2005. **30**(5): p. 70-79.
94. Li, H.C., S. Krishnamurthi, and K. Fisler. *Interfaces for Modular Feature Verification*. in *Proceedings of the 17 th IEEE International Conference on Automated Software Engineering (ASE'02)*. 2002.

95. Lin, L., B. Nuseibeh, D. Ince, and M. Jackson. *Using abuse frames to bound the scope of security problems*. in *Proceedings of 12th IEEE International Requirements Engineering Conference*. 2004.
96. Lin, L., B. Nuseibeh, D. Ince, M. Jackson, and J. Moffett. *Introducing abuse frames for analysing security requirements*. in *Proceedings of 11th IEEE International Requirements Engineering Conference*. 2003.
97. Lin, L., S.J. Prowell, and J.H. Poore, *The impact of requirements changes on specifications and state machines.* Software: Practice and Experience, 2009. **39**(6): p. 573-610.
98. Liu, L., E. Yu, and J. Mylopoulos. *Security and privacy requirements analysis within a social setting*. in *11th IEEE International Requirements Engineering Conference*. 2003.
99. Lodderstedt, T., D. Basin, and J. Doser, *SecureUML: A UML-Based Modeling Language for Model-Driven Security*, in *«UML» 2002: The Unified Modeling Language*. 2002. p. 426-441.
100. Lutz, R.R. and I.C. Mikulski, *Operational anomalies as a cause of safety-critical requirements evolution.* Journal of Systems and Software, 2003. **65**(2): p. 155-161.
101. Mantel, H. *On the composition of secure systems*. in *IEEE Symposium on Security and Privacy*. 2002.
102. Mantel, H. *Preserving information flow properties under refinement*. in *IEEE Symposium on Security and Privacy*. 2001.
103. Mead, N.R. and T. Stehney, *Security quality requirements engineering (SQUARE) methodology.* SIGSOFT Software Engineering Notes, 2005. **30**(4): p. 1-7.
104. Mens, K., T. Mens, and M. Wermelinger, *Supporting software evolution with intentional software views*, in *Proceedings of the International Workshop on Principles of Software Evolution*. 2002, ACM: Orlando, Florida. p. 138-142.
105. Mens, T., J.F. Ramil, and M.W. Godfrey, *Analyzing the Evolution of Large-Scale Software.* Journal of Software Maintenance and Evolution: Research and Practice, 2004. **16**(6): p. 363 - 365.
106. Mens, T., M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. *Challenges in software evolution*. in *8th International Workshop on Principles of Software Evolution*. 2005.
107. Mouratidis, H. and P. Giorgini, *Integrating Security and Software Engineering: Advances and Future Visions*. 2006, London, United Kingdom: Idea Group Publishing. 288.
108. Mouratidis, H., P. Giorgini, and G. Manson, *Modelling secure multiagent systems*, in *Proceedings of the 2nd international joint conference on Autonomous agents and multiagent systems*. 2003, ACM: Melbourne, Australia. p. 859-866.
109. Mouratidis, H., P. Giorgini, and G. Manson, *When security meets software engineering: a case of modelling secure information systems.* Information Systems, 2005. **30**(8): p. 609-629.
110. Mouratidis, H., J. Jurjens, and J. Fox, *Towards a Comprehensive Framework for Secure Systems Development*, in *Advanced Information Systems Engineering*. 2006. p. 48-62.
111. Nhlabatsi, A., R. Laney, and B. Nuseibeh, *Feature Interaction: the Security Threat from within Software Systems.* Progress in Informatics, 2008(5): p. 75-89.
112. Nuseibeh, B., S. Easterbrook, and A. Russo, *Leveraging Inconsistency in Software Development.* Computer, 2000. **33**(4): p. 24-29.
113. Opdahl, A.L. and G. Sindre, *Experimental comparison of attack trees and misuse cases for security threat identification.* Information and Software Technology, 2009. **51**(5): p. 916-932.
114. O'Reilly, C., P. Morrow, and D. Bustard. *Lightweight prevention of architectural erosion*. in *Proceedings of 6th International Workshop on Principles of Software Evolution*. 2003.
115. Oreizy, P., 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 and Their Applications, 1999. **14**(3): p. 54 - 62.
116. Palmer, S.R. and J.M. Felsing, *A Practical Guide to Feature-Driven Development*. 2002: Pearson Education.
117. Parsons, D., A. Rashid, A. Telea, and A. Speck, *An architectural pattern for designing component-based application frameworks.* Software: Practice and Experience, 2006. **36**(2): p. 157-190.
118. Pena, J., M.G. Hinchey, M. Resinas, R. Sterritt, and J.L. Rash, *Designing and managing evolving systems using a MAS product line approach.* Science of Computer Programming, 2007. **66**(1): p. 71-86.
119. Ramil, J.F. *Laws of software evolution and their empirical support*. in *International Conference on Software Maintenance*. 2002.
120. Ramil, J.F. and N. Smith, *Qualitative simulation of models of software evolution.* Software Process: Improvement and Practice, 2002. **7**(3-4): p. 95-112.
121. Rath, I., G. Bergmann, A. Okros, and D. Varro, *Live Model Transformations Driven by Incremental Pattern Matching*, in *Proceedings of the 1st international conference on Theory and Practice of Model Transformations*. 2008, Springer-Verlag: Zurich, Switzerland. p. 107-121.
122. Ravichandar, R., J.D. Arthur, and S.A. Bohner. *Capabilities Engineering: Constructing Change-Tolerant Systems*. in *40th Annual Hawaii International Conference on System Sciences*. 2007.

123. Ravichandar, R., J.D. Arthur, S.A. Bohner, and D.P. Tegarden, *Improving change tolerance through Capabilities-based design: an empirical analysis.* Journal of Software Maintenance and Evolution: Research and Practice, 2008. **20**(2): p. 135-170.
124. Ren, X., O.C. Chesley, and B.G. Ryder, *Identifying Failure Causes in Java Programs: An Application of Change Impact Analysis.* IEEE Transactions on Software Engineering, 2006. **32**(9): p. 718-732.
125. Ricci, A., *Agents and Coordination Artifacts for Feature Engineering*, in *Objects, Agents, Features*, J.-J.C.M. Mark D. Ryan, Hans-Dieter Ehrich, Editor. 2004, Springer-Verlag: Berlin Heidelberg. p. 209–226.
126. Roshandel, R., A.V.D. Hoek, M. Mikic-Rakic, and N. Medvidovic, *Mae - a system model and environment for managing architectural evolution.* ACM Transactions in Software Engineering Methodology, 2004. **13**(2): p. 240-276.
127. Russo, A., B. Nuseibeh, and J. Kramer, *Restructuring requirements specifications.* IEE Proceedings Software, 1999. **146**(1): p. 44-53.
128. Russo, A., B. Nuseibeh, and J. Kramer. *Restructuring requirements Specifications for Managing Inconsistency and Change: A Case Study*. in *Proc. of 3 rd International Conference on Requirements Engineering (ICRE `98)*. 1998. Colorado Springs, USA.
129. Rysselberghe, F.V. and S. Demeyer. *Studying Software Evolution Information by Visualizing the Change History*. in *Proceedings of the 20th IEEE International Conference on Software Maintenance*. 2004.
130. Salifu, M., Y. Yu, and B. Nuseibeh. *Specifying Monitoring and Switching Problems in Context*. in *Proceedings of the 15th IEEE International Conference in Requirements Engineering (RE '07)*. 2007. New Delhi, India.
131. Seybold, C., S. Meier, and M. Glinz. *Evolution of requirements models by simulation*. in *Procedings of 7th International Workshop on Principles of Software Evolution*. 2004.
132. Shin, M.E. and H. Gomaa, *Software requirements and architecture modeling for evolving non-secure applications into secure applications.* Science of Computer Programming, 2007. **66**(1): p. 60-70.
133. Sindre, G. and A.L. Opdahl. *Eliciting security requirements by misuse cases*. in *Proceedings of 37th International Conference on Technology of Object-Oriented Languages and Systems*. 2000.
134. Sindre, G. and A.L. Opdahl, *Eliciting security requirements with misuse cases.* Journal of Requirements Engineering, 2005. **10**(1): p. 34-44.
135. Smith, G. and T. McComb, *Refactoring Real-time Specifications.* Electronic Notes in Theoretical Computer Science, 2008. **214**: p. 359-380.
136. Smith, N., A. Capiluppi, and J.F. Ramil, *A study of open source software evolution data using qualitative simulation.* Software Process: Improvement and Practice, 2005. **10**(3): p. 287-300.
137. Soffer, P., *Scope analysis: identifying the impact of changes in business process models.* Software Process: Improvement and Practice, 2005. **10**(4): p. 393-402.
138. Tondel, I.A., M.G. Jaatun, and P.H. Meland, *Security Requirements for the Rest of Us: A Survey.* IEEE Software, 2008. **25**(1): p. 20-27.
139. Trujillo, J., E. Soler, E. Fernández-Medina, and M. Piattini, *An engineering process for developing Secure Data Warehouses.* Information and Software Technology, 2009. **51**(6): p. 1033-1051.
140. Turner, C.R., *Feature Engineering of Software Systems*, in *Computer Science*. 1999, University of Colorado. p. 175.
141. Turner, C.R., A. Fuggetta, L. Lavazza, and A.L. Wolf, *A Conceptual basis for feature engineering.* The Journal of Systems and Software, 1999. **49**(1): p. 3-15.
142. Turner, K.J. *An Architectural Foundation for Relating Features*. in *Proceedings of Feature Interactions in Telecommunication Networks IV*. 1997. Amsterdam: IOS Press.
143. van Lamsweerde, A. *Elaborating security requirements by construction of intentional anti-models*. in *26th International Conference on Software Engineering*. 2004.
144. van Lamsweerde, A., R. Darimont, and E. Letier, *Managing conflicts in goal-driven requirements engineering.* IEEE Transactions on Software Engineering, 1998. **24**(11): p. 908-926.
145. van Lamsweerde, A. and L. Willemet, *Inferring declarative requirements specifications from operational scenarios.* IEEE Transactions on Software Engineering, 1998. **24**(12): p. 1089-1114.
146. Velthuijsen, H., *Issues of non-monotonicity in feature interaction detection*, in *Feature Interactions in Telecommunication Systems III*, K.E. Cheng and T. Ohta, Editors. 1995, IOS Press: Amsterdam. p. 31-42.
147. Villarroel, R., E. Fernández-Medina, and M. Piattini, *Secure information systems development - a survey and comparison.* Computers & Security, 2005. **24**(4): p. 308-321.
148. Wang, Q., J. Shen, X. Wang, and H. Mei, *A component-based approach to online software evolution.* Journal of Software Maintenance and Evolution: Research and Practice, 2006. **18**(3): p. 181-205.
149. Wu, J., R.C. Holt, and A.E. Hassan, *Exploring Software Evolution Using Spectrographs.* 11th Working Conference on Reverse Engineering, 2004: p. 80-89.
150. Xu, D., V. Goel, and K. Nygard. *An Aspect-Oriented Approach to Security Requirements Analysis*. in *30th Annual International Computer Software and Applications Conference*. 2006.

151. Yokogawa, T., T. Tsuchiya, M. Nakamura, and T. Kikuno, *Feature Interaction Detection by Bounded Model Checking.* IEICE Transactions on Information and Systems, 2003. **E86-D**(12): p. 2579-2587.
152. Zave, P., *An experiment in feature engineering.* Programming methodology: Monographs In Computer Science, 2003: p. 353 - 377.
153. Zave, P. *Requirements for Evolving Systems: A Telecommunications Perspective*. in *Proceedings of 5th IEEE International Symposium on Requirements Engineering (RE'01)*. 2001. Toronto, Canada: IEEE Computer Society.
154. Zave, P., H.M. Goguen, and T.M. Smith, *Component Coordination: A telecommunication case study.* Computer Networks: The International Journal of Computer and Telecommunications Networking, 2004. **45**(5): p. 645-664.
155. Zave, P. and M. Jackson, *Four dark corners of requirements engineering.* ACM Transactions on Software Engineering and Methodology, 1997. **6**(1): p. 1-30.
156. Zenger, M., *KERIS: evolving software with extensible modules.* Journal of Software Maintenance and Evolution: Research and Practice, 2005. **17**(5): p. 333-362.
157. Zowghi, D. and R. Offen. *A logical framework for modeling and reasoning about the evolution of requirements*. in *3rd IEEE International Symposium on Requirements Engineering*. 1997.

APPENDIX C – Evolution in Access Control Systems

Evolution in Access Control Systems: A State of the Art Survey

Federica Paci and Fabio Massacci

Department of Information Engineering and Computer Science, University of Trento

Abstract. Access control is a key component of system security. Due to the high dynamicity that characterize the new software systems, access control models must be able to evolve with the systems they protect to maintain an adequate level of security. This survey discusses what does it mean evolution in access control and provides an overview of the access control models that are able to support evolution.

1 Introduction

Access control is a key component in any security solution, in that it determines which *subject* can perform which *action* under which circumstances on the protected *resources*.

A variety of access control models and policy languages have been developed over the years. They can be classified according to three main access control paradigms: Discretionary Access Control (DAC), Mandatory Access Control (MAC), and Role Based Access Control (RBAC).

In the early days of the mainframe, when the biggest need was to prevent one user from interfering with the work of others sharing the machine, MAC models were widely adopted. Permissions to use a system resource, such as a file, was linked to the users identity. Permissions were stored in an access matrix, that can be modified only by a trusted party, the system administrator. As the number of users grew, the burden on the administrator became untenable. Thus, DAC models have emerged as a more suitable access control mechanisms. In DAC models each object has an owner who exercises primary control over the object. While DAC performed well for centralized monolithic systems, it turned not to be suitable for distributed systems because of the complexity of managing the access rights for individuals and machines. Role-based access control (RBAC) proposed to reduce such administrative costs. In RBAC permissions are assigned to roles, that identify a category of users, rather then to single users. The idea is that there will be far fewer roles than either users or permissions.

Such systems where designed with an essentially static idea in mind: the users and the resources change slowly and the management of changes is essentially a problem of administration. Indeed, the classical Bell-LaPadula model for multi-level security introduced the "tranquility principle" as a key property for security theorems to hold [6].

Unfortunately, new software systems are undergoing continuing change and rapid revolution to respond to the changes in the environment, user needs, developing concepts and advancing technologies [17].

So, the evolution of software systems requires access control systems that include evolution as a first class citizen in order to preserve the security of such systems.

With the exception of the early papers on the access control matric by Harrison, Ruzzo and Ullmann [11] (and the first works by Sandhu [22, 23]), the problem of evolution of access control systems has received little attention by the research community because was considered just a problem of administration. This survey provides an overview of how access control models deal with evolution.

The survey is structured as follows. In the next section, we discuss what evolution means in access control systems and which are the main causes of evolution. Section 3 overviews the proposals that supports the representation of change and policies change impact analysis. Section 4 presents access control models that consider environmental factors in making access control decisions. Section 5 discusses the only work about resiliency of an access control model. Moreover, Section 6 concludes the survey and outlines some possible research directions.

2 The Notion of Evolution in Access Control

The evolution of an access control model consists of changing the access control policies and the constraints on the existing model. The possible type of changes that can occur in an access control model consists of adding or removing a subject, adding or removing a resource to be protected, and in adding, revoking or modifying access rights granted to subjects. These changes can be triggered by several factors. Access control policies may change because of *changes in the environment* in which the access control model is deployed. For example, when the system is attacked, or when a vulnerability is detected, the access rights granted to users need to be changed to prevent malicious users or software applications from accessing vulnerable resources, or vice-versa. When new regulations or laws concerning security and privacy are introduced, access control policies must be updated to comply with new regulations and laws. The access control policies that regulate the access to a resource may also change with environmental factors such as time or the location in which users make an access request. *Changes of the system requirements* may also cause the modification of access control policies. Indeed, system requirements and access control policies are strictly related. Requirements capture the functionalities of a system while access control policies control end-users interactions with system resources that are usually described by functional requirements.

Changes to the system design and implementation such as the addition of a software component, or of a new resource, the installation or the update of new application requires the specification of new access control policies to restrict access to the new resources.

Thus, it is important that an access control model is able to evolve in response to the variety of changes that can occur and undermine the security of a system.

An access control model should support the representation of the events that cause the evolution of the model and the analysis of the impact of the change in terms of the results of the evaluation of access requests with the respect to the policies and of the authorization state that lead to a specific evaluation result.

Though evolution of access control models is inevitable to preserve systems' security, how to manage the evolution of an access control model is still an open problem. The proposals about access control models and evolution can be classified in proposals that:

1. support the representation of change
2. support change impact analysis
3. evaluate the impact of changes in the environment on the applicability of access control policies
4. analyze the resiliency of access control models to changes.

In the next sections, we present the main proposals about evolution of access control models based on the above classification.

3 Evolution as Change in Policies

Analyzing what have caused a change in the access control policies of a system and how this change affects the set of actions that are permitted or denied is really important. In fact, changes in the policies may result in a decreased level of protection.

Margrave [14] is a software suite for analyzing the impact of changes to role-based access-control policies expressed in XACML [19]. Margrave includes a verifier that analyzes policies written in the XACML language, translating them into MTBDDs (multi-terminal binary decision diagrams). The vertexes of the diagram model variables that represent the components subjects, actions, and resources of an XACML policy. Each combination of boolean values over these variables maps to one of three policy results (permit, deny, or not-applicable) supported by XACML; the results are denoted by the terminals of the MTBDD. To implement change analysis, Margrave introduces a different type of MTBDD called change-analysis decision diagram or CMTBDD. A CMTBDD has sixteen terminals, one for each ordered pair of results from the policies being compared (such as permit-to-permit, deny-to-ec, permit-to-not-applicable, and so on). The CMTBDD is generated from the MTBDDs of the two polices need to be composed showing the changes of the two policies. Margrave provides a suite of operators for creating an manipulating a CMTBDD, such as restricting a CMTBDD to a particular kind of change and determining which variable values can lead to particular kinds of changes.

Pucella and Weissman [21] introduce a modal logic-based on propositional dynamic logic to reason about the execution of scenarios during which the set of access control policies change. The semantic of the logic is based on Kripke structures, which are the formal models of the applications. Intuitively, a Kripke structure encodes a transition system, along with the characteristics of each state

(i.e., which primitive propositions are true in each state). Transitions represent the actions that can be permitted or denied. To analyze the consequences of changing an access control policy, the authors model the properties that the change should satisfy as a formula, and verifies that the formula is true with respect to the Kripke structure capturing the states of the application and the possible transitions, and the new set of access control policies.

Koch, Mancini, and Parisi-Persicce study the change impact problem [15]. They use graph transformations to represent the evolution, the integration, and the transformation of security policies. A policy is formalized by four components: a type graph, positive and negative constraints (a declarative way of describing what is wanted and what is forbidden) and a set of rules (an operational way of describing what can be constructed). The specification formalism, is based on the different possible semantics of graph transformation systems, described in terms of category theory and well understood gluing constructions. They discuss how to preserve the coherence of a policy during its evolution. They assume that the change over time of a policy is due to the addition/deletion of rules and constraints. Although they present examples of how to represent changes using graphs, they present no algorithms or tools, nor suggest methods for eliciting policy change from graph differences.

Chaudhuri et al. [7] propose EON, a logic-programming language to model and automatically analyze dynamic access control systems. The authors focus on access control systems in which processes and objects are labeled with security levels, and processes are prevented to access objects based on their labels. The changes that the author consider and of which they analyze the impact are the creation of new objects and processes, and the modification of objects and security labels. Thus, EON language extends Datalog with dynamic operators for creating and modifying simple objects and processes. The operational semantic of an EON program, that is, a collection of clauses, is given as a (possible non deterministic) transition system over a database, that is a collection of facts. The analysis of the access control system is done evaluating a query on an EON program and checking that the system does not reach an undesirable state.

Naldurg and Campbell [20] present an approach to dynamically changing access rights in response to an attack to the system or when a vulnerability is detected so that the safety property and trust assumptions are preserved. The approach is based on representing the possible changes as a state machine with the sets of subjects, objects and access rights as its state variables, and the transitions are all system actions that can change the state variables. State-changing transitions may include the addition or removal of a subject, an object, or an access right. To preserve the safety property and trust assumptions, transitions are associated with guards that force users to present a proof of authorization, in the form of credentials, attesting that they have the right to change the access rights.

Barker et al. [1] present SBAC, a novel access control model based on the notion of status. The key aspect of the SBAC model is the capability of autonomously changing access control policies in response to events that involve

users actions. The assignment of users and access privileges on objects may change dynamically as a consequence of the occurrence of events of relevance in an environment being modeled or because of situational factors, such as the time at which access to a resource is requested, the location of the agent requesting access, CPU or network load measures, system status criteria (e.g., system under attack), sales volumes, and trading patterns. These events are used, in conjunction with users action status, to determine a users status level and hence the users authorizations. A users action status; this history enables changing access policy requirements to be naturally accommodated. SBAC access control policies and the history of events related to a user are represented as Identification-based Logic Programs (IBLPs), that are an annotated form of logic programs. The approach is implemented as an autonomous agent that reasons about the events, actions, and a history (of events and actions), which relates to a requester for access to resources, in order to decide whether the requester is permitted the access to a resource.

4 Evolution as Change in the Environment

The importance of taking environmental factors into account when making access control decision has been recently outlined in several proposals [10, 9, 13, 2, 12, 24, 25, 8, 5, 26, 16, 4, 3]. Environmental factors such as time and location are often denoted as context in these proposals. Considering environmental factors allows to make access control policies enforcement dynamic. Changes in the environment influence the applicability of access control policies, and trigger the dynamic change of the policies.

Dougherty et al. [10] define a framework to represent access control policies, their dynamic environment and the interactions between them. A policy interacts with its dynamic environment by consulting facts in the environment and potentially constraining certain actions in the environment. The interaction between a policy and its environment is modeled by a state machine. States are labeled with a set environmental facts and the result of the evaluation of the policy in that state, while transitions are labeled with events corresponding to actions performed by users (such as access requests) and events occurring in the environment. Such model is used to analyze the impact of the environment on the evaluation of an access request against an access control policy.

Craven et al. [9] present an expressive logical framework for policy specification and analysis. The framework separates the representation of policies from the representation of the system that is protected by the policies. Policies are represented as first-order logic rules while to model the system Event Calculus is adopted. EC has been chosen because of its ability to represent concisely the effect of actions on properties of the system. EC includes predicates to represent dynamic features of the system, system events not regulated by policies, system events regulated by policies and time instants. The analysis is based on the use of abductive, constraint logic programming (ACLP) systems, and the Event Calculus (EC) to describe how events and actions occurring in the system

affect the system states, leading to circumstances in which a given policy rule is applicable. The analysis returns a system trace, that is sequence of actions, that have caused a change in system properties.

Jagadeesan et al. [13] propose a policy algebra for dynamic policies that is a sub-language of Time Default Concurrent Constraint programming. The programs in their policy algebra are reactive, meaning that a program may interact with its environment in a sequence of discrete time steps. With this algebra it is possible to represent state changes triggered by environmental changes or users access requests. State changes are modeled using labeled transition systems. Each state of the LTS is captured by a Datalog constraint program. An LTL formula is interpreted over traces (sequences of states) where in each state of the trace, a truth value is associated with each of the atoms appearing in the formula. LTL has temporal operators in addition to the usual logical connectives of propositional logic so that one can describe relationships between the values of the atoms across time.

Becker et al. [2] propose SMP, a logic for specifying access control policies whose evaluation causes a change in the authorization state. This logic is based on Datalog, but extends it with predicates for state modification, called effects, and a simple form of negation. The semantics of SMP is formalized by modeling it as a fragment of Transaction Logic that is a general framework that incorporates database updates and transactions into first order logic. An authorization state is defined by a database which contains environmental facts that are relevant for authorization, such as the actions a user performs or the role played by a user. Facts may be inserted or removed from the database as the result of evaluating an access request. The authors present an inference system for evaluating sequences of user actions with respect to policies and check that the authorization state reached satisfies certain constraints. Thus, they do not consider changes to the policies but only changes in the environmental facts.

Hulsebosch et al. [12] propose a context-based access control model where the access to a resource is granted or denied to a user based on context information. The introduction of the context allows to adapt security policies to situational or contextual changes. They propose a system architecture to take authorization decision based on context that consists of context owners (COs), context brokers (CBs), context providers (CP), context-aware service providers (CASPs), and access controllers (ACs). The COs collect and own the contextual data or information, e.g. a user receives and possesses GPS-location information. He or she decides how and by whom context data or information may be stored, distributed and processed. The CPs check that context access control and usage policies are in line with privacy and security requirements of the COs. The CPs also take care of context management issues by providing categorization means for context indexing, retrieval, querying, inferential and association purposes. CBs provide service publishing mechanisms to CPs, and service discovery mechanisms to the CASPs. The CASPs provide services to users that are adapted to specific user-service contexts, e.g. being on the train. The AC grants or denies users to perform an operation on an object according to an access control policy.

First, the AC authenticates a user by verifying the contextual attributes provided by the user. Then, the AC binds the user with a set of permissions based on those attributes.

Toahchoodee et al. [24] present Spatio-Temporal RBAC (STRBAC), which extends traditional RBAC models with spatial and temporal constraints. In STRBAC, users assignment to roles and permissions assignment to roles is based on the location of a user and on a time interval. Also, the enforcement of separation of duty constraints depends on location and time. Thus, the permissions of a user change over time and depending on the user location. To analyze STRBAC and the system that it protects, the authors propose an approach based on using UML and OCL language to model STRBAC components and the system. The UML model is, then, translated into an Alloy [1] specification that can be automatically analyzed using the Alloy Analyzer which has embedded SAT-solvers. The results of the analysis indicate the level of protection provided by the STRBAC model for the given system.

In [25], Toninelli et al. a semantic context-aware access control model. The context includes information about the resources accessed, the actors and the surrounding environment, and it is modeled in Web Ontology Language (OWL). The policies life cycle consists of three distinct phases: *policy specification*, *policy refinement*, and *policy evaluation*. In the policy specification phase resource administrators specify OWL-based policies representing ontological associations between actions and contexts ontology definitions. The protection contexts may have attribute values assigned to constants or may be variables. In the latter case, attributes are assigned proper values by combining DL-based and LP-based reasoning over the context ontology and a set of context aggregation and activation rules. In particular, the output of LP rules is fed into the DL knowledge base to determine the value of each attribute given the current context. OWL-based policies can be viewed as policy types: they define the actions that are allowed in a set of context types. In the policy specification phase, administrators have to define aggregation and evaluation rules to enable effective enforcement and adaptation of OWL policies. In order to be enforced in the real world, policies need to be instantiated by adapting them to the current state of the world, in order to obtain the set of applicable policies. Once the set of applicable policies is determined, the contexts of applicable policies are verified against the current state of context elements as measured by sensors to determine the set of currently active policies.

In [8], Convington et al. extend RBAC model with a new type of role called environment role. Environment roles capture relevant environmental conditions that are used for restricting and regulating user privileges. Environment are active when the conditions that define the roles are satisfied in the current environmental state. Accesses to resources are granted to users if users are assigned to a role who has the permission to access the resources and to an environment role that is active at the moment of the access request is submitted. Clearly, per-

[1] Alloy is a is a fully declarative first-order logic language designed for modeling and analyzing complex software systems.

missions may change for a single users accessing a resource if the environmental conditions vary between requests.

Zhang et al. [26] propose Dynamic Role Based Access Control (DRBAC) another access control model that extends RBAC with context information. Context information includes environment of the user such as location, time that the user access the resource and system information such as CPU usage and network bandwidth. The privileges that users have changes based on such context information. The possible roles which can be assigned to a user and the permissions that are assigned to a role are modeled as state machines. The transitions represent the events that trigger the change in the assignment of roles to users and the assignment of permissions to roles. Transitions trigger because an event occur which is generated by a Context Agent in response to a change in context information. Thus, Zhang et al. provide a way to represent the changes that can occur in their model but they do not propose an approach to analyze the impact of such changes.

Bertino et al. [5] present GEO-RBAC, an extension of RBAC model to deal with spatial and location-based information. GEO-RBAC relies on the OGC spatial model to represent (spatial) objects, user positions, and geographically bounded roles, making the approach quite standard and flexible. Another important characteristic of the model is the ability to deal with either real positions, obtained from a given mobile terminal or a cellular phone, and logical ones, possibly represented at different granularities. GEO-RBAC is based on the notions of features, role schema and spatial role. Features are entities of the real world that may occupy a position and are characterized by a type. A role schema defines some common properties of a set of spatially aware organizational functions with a similar meaning. A role schema not only defines a common name for a set of spatial roles but also constrains the space where roles can be enabled, the so called *role extent*. Moreover it specifies the type of logical locations and ultimately the granularity of the position that the users playing that role may occupy. A spatial role is a role schema instance. Users are assigned spatial roles, that can be activated during a session. Unlike RBAC, roles are enabled only when the user position is contained in the role extent. Finally, like RBAC, GEO-RBAC supports the notion of role hierarchy that allows a role to inherit permissions from its ancestor roles, users from its descendant roles, and roles to be enabled when descendant roles are.

In [16] Kulkarni and Tripathi present CA-RBAC, a context-aware RBAC model that extends RBAC in several directions. The model supports personalized permissions for role members, and context-based constraint specification as part of - dynamic binding of objects with active space services, user admission to roles, permission executions by role members, and granting access to a subset of a services resources based on a role members context information. The model also supports revocation of a users membership in a role when context conditions fail to hold. Based on this model, the authors have developed a role-based framework for programming secure context-aware pervasive computing applications.

Bertino et al. [4] propose Temporal-RBAC (TRBAC), an extension of RBAC

models that supports temporal constraints on the enabling/ disabling of roles. TRBAC supports periodic role enabling and disabling, and temporal dependencies among such actions. Such dependencies expressed by means of role triggers (active rules that are automatically executed when the specified actions occur) can also be used to constrain the set of roles that a particular user can activate at a given time instant. The firing of a trigger may cause a role to be enabled/disabled either immediately, or after an explicitly specified amount of time. Enabling/disabling actions may be given a priority that may help in solving conflicts, such as the simultaneous enabling and disabling of a role. They also propose a polynomial algorithm to verify whether TRBAC specifications are safe that is they are free from ambiguities.

In another work, Bertino et al. [3] present an access control model in which periodic temporal intervals are associated with authorizations. Permissions are often limited in time or may hold only for specific periods of time. An authorization is automatically granted in the specified intervals and revoked when such intervals expire. Deductive temporal rules with periodicity and order constraints are provided to derive new authorizations based on the presence or absence of other authorizations in specific periods of time.

5 Resiliency to Change

Another aspect related to the evolution of an access control system is the resiliency to absence of users.

In [18], Li and Wang introduce a new type of policies denoted as *resiliency policies* in the context of access control systems. Resiliency policies state properties about enabling access in an access control system rather than restricting access as access control policies do. Intuitively, a resiliency policy specifies a fault tolerance requirement with respect to a certain critical task. A resiliency policy consists of the set of permissions that are needed to carry out the task, the number of absent users the system should tolerate, and the number of the disjoint sets of users such that the users in each set together possess the permissions to perform the task. The authors discusses the Resiliency Checking Problem that consists in determining whether an access control state satisfies a given resiliency policy. In the general case such problem and several sub cases are intractable (NP-hard), but the authors identify two sub cases that are solvable in linear time.

6 Conclusions

In this survey we have discussed the evolution of access control models and how it is related to the evolution of software systems. In particular, we have first identified the causes of the evolution of an access control model. Then, we have provided an overview of the main proposals about the evolution of access control models. From the analysis of these proposals, we can conclude that to analyze

access control models evolution there is the need for a policy framework based on a first-order logic whose semantics can be represented as state machines. State machines are suitable for representing the events that trigger the evolution of access control policies. Moreover, most of the proposals deal only with changes in the authorization models caused by changes in the environment. Policy evolution due to changes at requirements, design and implementation level has not been investing. Thus, analyzing the dependencies between evolution of access control models and evolution of requirements, design and implementation is an interesting future research direction.

References

1. Steve Barker, Marek J. Sergot, and Duminda Wijesekera. Status-based access control. *ACM Trans. Inf. Syst. Secur.*, 12(1):1–47, 2008.
2. Moritz Y. Becker and Sebastian Nanz. A logic for state-modifying authorization policies. In *ESORICS*, pages 203–218, 2007.
3. Elisa Bertino, Claudio Bettini, Elena Ferrari, and Pierangela Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM Trans. Database Syst.*, 23(3):231–285, 1998.
4. Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. Trbac: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, 2001.
5. Elisa Bertino, Barbara Catania, Maria Luisa Damiani, and Paolo Perlasca. Geo-rbac: a spatially aware rbac. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 29–37, New York, NY, USA, 2005. ACM.
6. Matt Bishop. *Computer Security.* Addison Wesley, 2003.
7. Avik Chaudhuri, Prasad Naldurg, Sriram Rajamani, Ganesan Ramalingam, and Lakshmisubrahmanyam Velaga. Eon: Modeling and analyzing dynamic access control systems. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, pages 381–390. ACM, 2008.
8. Michael J. Covington, Wende Long, Srividhya Srinivasan, Anind K. Dev, Mustaque Ahamad, and Gregory D. Abowd. Securing context-aware applications using environment roles. In *SACMAT '01: Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 10–20, New York, NY, USA, 2001. ACM.
9. Robert Craven, Jorge Lobo, Jiefei Ma, Alessandra Russo, Emil Lupu, and Arosha Bandara. Expressive policy analysis with enhanced system dynamicity. In *ASI-ACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 239–250, New York, NY, USA, 2009. ACM.
10. Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *of Lecture Notes in Computer Science*, pages 632–646. Springer, 2006.
11. M.H. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, 1976.
12. R. J. Hulsebosch, A. H. Salden, M. S. Bargh, P. W. G. Ebben, and J. Reitsma. Context sensitive access control. In *SACMAT '05: Proceedings of the tenth ACM*

symposium on Access control models and technologies, pages 111–119, New York, NY, USA, 2005. ACM Press.
13. Radha Jagadeesan, Will Marrero, Corin Pitcher, and Vijay A. Saraswat. Timed constraint programming: a declarative approach to usage control. In *PPDP*, pages 164–175, 2005.
14. Fisler K., Krishnamurthi S., Meyerovich L. A, and Tschantz M. C. Verification and change-impact analysis of access control policies. In *International Conference on Software Engineering (ICSE)*, 15-21 May 2005.
15. M. Koch, L. V. Mancini, and F. Parisi-Presicce. On the specification and evolution of access control policies. In *Symposium on Access Control Models and Technologies (SACMAT)*, 3-4 May 2001.
16. Devdatta Kulkarni and Anand Tripathi. Context-aware role-based access control in pervasive computing systems. In *SACMAT*, pages 113–122, 2008.
17. Meir M. Lehman and Juan F. Ramil. Software evolution: background, theory, practice. *Inf. Process. Lett.*, 88(1-2):33–44, 2003.
18. Ninghui Li, Qihua Wang, and Mahesh Tripunitara. Resiliency policies in access control. *ACM Trans. Inf. Syst. Secur.*, 12(4):1–34, 2009.
19. Tim Moses. extensible access control markup language tc v2.0 (xacml), February 2005.
20. Prasad Naldurg and Roy H. Campbell. Dynamic access control: preserving safety and trust for network defense operations. In *SACMAT*, pages 231–237, 2003.
21. Pucella R. and Weissman V. Reasoning about dynamic policies. In *Foundations of Software Science and Computation Structures (FOSSACS)*, 2004.
22. Ravi Sandhu. The typed access matrix model. pages 122–136, 1992.
23. Ravi Sandhu and Srinivas Ganta. On testing for the absence of right in access control models. pages 109–118, 1993.
24. Manachai Toahchoodee, Indrakshi Ray, Kyriakos Anastasakis, Geri Georg, and Behzad Bordbar. Ensuring spatio-temporal access control for real-world applications. In *SACMAT '09: Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 13–22, New York, NY, USA, 2009. ACM.
25. Alessandra Toninelli, Rebecca Montanari, Lalana Kagal, and Ora Lassila. A semantic context-aware access control framework for secure collaborations in pervasive computing environments. In *International Semantic Web Conference*, pages 473–486, 2006.
26. Guangsen Zhang and Manish Parashar. Context-aware dynamic access control for pervasive applications. In *Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS'04)*, 2004.