

# Requirements Evolution and What (Research) to Do about It

Neil A. Ernst<sup>1</sup>, John Mylopoulos<sup>1,2</sup>, and Yiqiao Wang<sup>1</sup>

<sup>1</sup> University of Toronto, Dept. of Computer Science, Toronto, Canada  
{`nernst,jm,yw`}@`cs.toronto.edu`

<sup>2</sup> 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<sup>1</sup>, 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.

---

<sup>1</sup> . . . with the notable exception of research on traceability mechanisms. Of course, traceability is useful for evolving requirements, but doesn’t actually solve the problem.

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 autonomous 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, Gırba 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 Pericone [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, meta-model, 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 constitute 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 classification 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.

Wiegiers [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 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 automatic 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 NK 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.**<sup>2</sup> 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 ([39, 40]) 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.

---

<sup>2</sup> . . . 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  $G_1, \dots, G_n$ , 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.

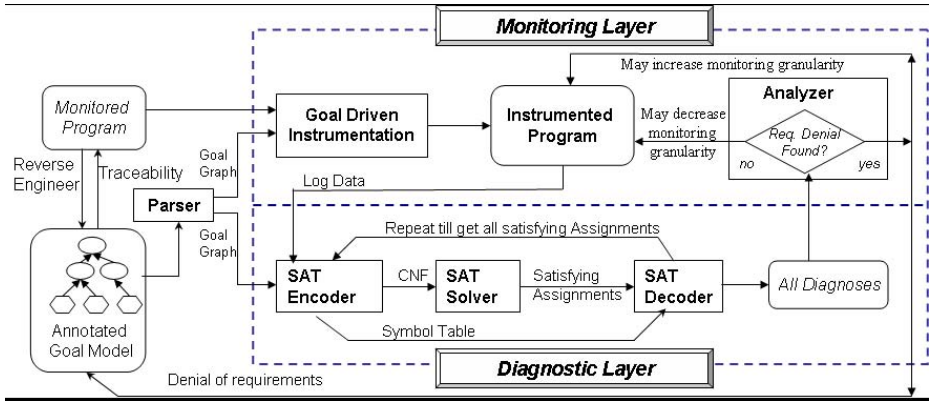


Fig. 1. Framework Overview

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. 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 session  $s$  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  $\Phi$ , where  $\Phi$  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)<sup>3</sup> 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 ended at timestep 2. In either of these cases, the diagnostic component infers  $FD(g, 3)$ , indicating that goal  $g$  is denied at time step 3.

---

<sup>3</sup> 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  $f$  remains 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  $g_1, \dots, g_n$ , and tasks  $a_1, \dots, a_m$ , 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:  $++S$ ,  $--S$ ,  $++D$ ,  $--D$ ,  $++$ ,  $--$ . Link  $++$  and link  $--$  are shorthand for the  $++S$  and  $++D$ , and the  $--S$  and  $--D$  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  $++D$  and  $--D$  are similar to those of  $++S$  and  $--S$ . 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  $D \cup \Phi$  ( $D \cup \Phi$ ) 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 [2].

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  $\Phi$ , 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  $\Phi$ . 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  $\Phi$  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  $\Phi$  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

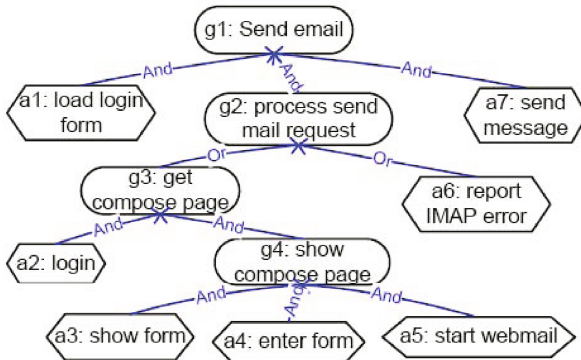


Fig. 2. Squirrel Mail Goal Model

**Table 1.** Squirrel Mail Annotated Goal Model

Goal/ Task	Monitor switch	Precondition	Effect
a1	on	correctURL entered	login form loaded
a2	on	$\neg$ wrongIMAP $\wedge$ login form loaded	(user logged in $\wedge$ correct pin) $\vee$ ( $\neg$ user logged in $\wedge$ $\neg$ 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 $\vee$ error reported
g2	off	login form loaded $\vee$ wrongIMAP	webmail started $\vee$ error reported
g3	off	login form loaded $\wedge$ $\neg$ 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  $g_4$  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:

*correct URL entered*(1), *occ<sub>a</sub>*( $a_1$ , 2), *login form loaded*(3),  *$\neg$ wrongIMAP* (4), *occ<sub>a</sub>*( $a_2$ , 5), *correct pin*(6), *user logged in*(6), *occ<sub>a</sub>*( $a_3$ , 7), *occ<sub>a</sub>*( $a_4$ , 8), *occ<sub>a</sub>*( $a_5$ , 9),  *$\neg$ webmail started*(10), *occ<sub>a</sub>*( $a_7$ , 11),  *$\neg$ email sent*(12).

The log data contains two errors ( *$\neg$ webmail started*(10), and *occ<sub>a</sub>*( $a_7$ , 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  $a_5$ :



## 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  $\Phi$ ; (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  $\Phi$ ; (2) time taken by the SAT solver to solve  $\Phi$  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  $\Phi$  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 differencing. 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.: Defining 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., Amele, R.: An examination of the effects 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] Wieggers, K.E.: Automating requirements management. *Software Development 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., Settini, 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., Settini, R., Zou, X., Solc, P.: Automated classification 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: Bellahsene, 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 Gorp, 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 tropes. 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/>