

# Evolution in requirements models

*Depth Oral Report*

Neil A. Ernst  
Department of Computer Science  
University of Toronto  
`nernst@cs.toronto.edu`

June 29, 2009

## **Abstract**

This literature review, part of the requirements for the qualifying oral exam milestone of the Ph.D. program, surveys papers concerning the topic of evolution in requirements models. The domain is requirements for software-based systems. The review begins with context-setting descriptions of the wider area of software maintenance and evolution. I present a taxonomy of terms in software evolution and requirements, and this leads into a discussion of the historical development of the field of evolution of requirements models.

In the bulk of the paper, I examine current themes in the area, including formal techniques, goal model approaches, and requirements management. I also examine other fields of study that might be relevant, including complex systems engineering, flow models, databases, and ontologies.

To conclude, I present a preliminary outline of my proposed research directions in this area.

# Contents

<b>1</b>	<b>Overview</b>	<b>1</b>
1.1	Outline of the paper . . . . .	1
<b>2</b>	<b>Context: software evolution and maintenance</b>	<b>1</b>
<b>3</b>	<b>Theories of evolution</b>	<b>2</b>
3.1	Neo-Darwinist evolutionary theory . . . . .	2
3.2	Other evolutionary proposals . . . . .	2
3.3	Applications to socio-technical systems . . . . .	3
<b>4</b>	<b>Key concepts in requirements evolution</b>	<b>5</b>
4.1	Background terms . . . . .	5
4.2	Evolution taxonomies and frameworks . . . . .	7
4.2.1	The nature of change . . . . .	7
4.2.2	Managing change . . . . .	8
4.2.3	Requirements evolution taxonomies . . . . .	9
<b>5</b>	<b>Historical perspectives</b>	<b>9</b>
5.1	SADT . . . . .	9
5.2	Conceptual Information Modeling . . . . .	10
5.3	ERAE . . . . .	10
5.4	RML . . . . .	11
5.5	Modeling information systems with Telos . . . . .	11
5.6	The NFR framework . . . . .	11
5.7	i* . . . . .	12
5.8	KAOS . . . . .	12
<b>6</b>	<b>Current themes</b>	<b>12</b>
6.1	Formal approaches . . . . .	13
6.1.1	Defeasible requirements models . . . . .	13
6.1.2	Specifying system properties . . . . .	14
6.1.3	Explicit language . . . . .	14
6.2	Semi-formal inconsistency handling in requirements evolution . . . . .	14
6.3	Goals, features, non-functional requirements, and aspects . . . . .	15
6.3.1	Features . . . . .	16
6.3.2	Aspects . . . . .	16
6.3.3	Goals and NFRs . . . . .	16
6.3.4	Variability . . . . .	17
6.4	Requirements management . . . . .	17
6.4.1	COTS and components . . . . .	18
6.4.2	Traceability . . . . .	18
6.5	Agile modeling . . . . .	19
<b>7</b>	<b>Empirical studies of evolution in requirements models</b>	<b>20</b>
<b>8</b>	<b>Other forms of requirements evolution</b>	<b>21</b>
8.1	Complex systems models . . . . .	21
8.2	Flow models . . . . .	22
8.3	Database schemas . . . . .	22
8.4	Ontology evolution . . . . .	23
<b>9</b>	<b>Research opportunities</b>	<b>23</b>

# 1 Overview

This paper presents a literature review of the field of requirements evolution, where requirements are seen as one application of conceptual modeling. Its scope is limited to an area where the relevant research papers number in the dozens, rather than hundreds.

Why is requirements evolution important? Increasingly software is not ‘decommissioned’, but rather updated, refined or altered, because the existing system is reliable, secure, and well-understood. Or more cynically, perhaps the cost to commission a brand-new system is overwhelming.

As well, there is a recent trend towards the use of software-as-service. A company could have multiple service offerings, with external users who depend on it. This type of software isn’t likely to ever be decommissioned. The physical implementation – code, servers, components – might be completely different, but the external requirements the service addresses, and features that service provides, continue to exist.

One of the assumptions of this paper is that requirements engineering is important throughout a system’s lifecycle (and informs development of different systems, as well). Persisting throughout the lifecycle mandates an ability to update, adapt and evolve the system’s requirements. Requirements might be defined in this sense as the art of describing the difference between the world-as-is and the world-to-be. At each phase of system development, requirements models provide a way to determine how much progress has been made towards the world-to-be.

Requirements may be intangible and tacit, or semi-formal and loosely defined, or specified formally over hundreds of pages. These are all forms of conceptual models. Fundamentally, a conceptual model is an abstraction of reality humans construct to make sense of that reality. They have varied uses and goals, some of which include problem-solving, sense-making, communication, and adaptation.

Well-constructed models are robust and can adapt to changing realities. Everything exists in a specific time period, and within a context of events. This suggests that support for evolution in both the syntax (instance, model, and meta-model) and semantics (model meaning) is important in a modeling formalism.

Requirements models capture the state of a system (indicative properties) and the desired state of a system-to-be (optative properties). I rely on the terminology of Jackson (1997) throughout this work to distinguish between the various models. There are many different means of constructing these models, including disagreement on how much external context is necessary.

## 1.1 Outline of the paper

I begin by assuming familiarity with basic notions in conceptual modeling and focus on how evolution is managed and integrated in the various frameworks and methodologies covered. For example, rather than discussing the Telos framework in depth, I look at evolution support in the framework.

The paper begins with a discussion of context, specifically, what is *not* included in the review. This is challenging, as requirements are integrated deeply in all facets of the software engineering process. I first examine standard notions of evolution, and discuss applications to software-based systems. To define terms used in the paper, I then present a taxonomy for requirements model evolution, including a survey of existing taxonomies. Following that, I survey historical approaches to requirements modeling. I look at some seminal early work, and examine the role of model evolution. Next, I turn to current themes in requirements model evolution. These include, among other things, logical approaches, goal analysis, agile modeling, and a look at some empirical studies. The next section takes a look at evolution in other modeling frameworks, such as workflows, systems, and databases. To provide direction for future research, the paper concludes with a look at some research opportunities.

## 2 Context: software evolution and maintenance

It is clear that in research, most topics are interconnected to some extent. Nonetheless, the scope of the paper must be manageable, and this implies some relevant information may be omitted.

The topic of evolving requirements models fits more generally into the topics of software evolution and maintenance. This discipline addresses the entire software lifecycle, from requirements, design, and

specification, to implementation and maintenance. How do requirements fit in? “[Empirical] studies show that the incorporation of new user requirements is the core problem for software evolution and maintenance (Bennett and Rajlich, 2000, p. 74).”

Requirements are involved in non-software systems as well. Requirements also overlap various ‘phases’ of the software development lifecycle. Changes in requirements occur and impact all stages of development, from modeling, to design, to implementation, to maintenance. Thus, while for software to evolve necessarily implies a change in requirements (and ultimately, a change in the environment), evolving requirements don’t necessarily imply changing software.

Here I list some of the topics that I haven’t examined in detail:

- It may be useful to maintain several different models and then re-integrate them at a later date. Model merging is not covered in this review.
- Verifying existing requirements models is important to ensure fit between model and implementation. I don’t examine verification and validation in this context.
- Software maintenance examines the management of existing software, change assessment, impact costs, etc. The results of maintenance activities – such as code refactoring – have large impacts on pre-existing requirements models. Often, these models aren’t updated accordingly. I touch on this issue, but largely assume that the maintenance activity has correctly identified the required changes, and that requirements models are involved in the activity.

For more information, Mens and Eden (2005) provide a good introduction to the various aspects of evolution involved in software engineering.

### 3 Theories of evolution

Are there parallels between requirements evolution and biological evolution? There seem to be some similarities. This section identifies some of the core notions in biological evolutionary theory, along with some newer theories about existing inconsistencies. I then suggest some formulations of these theories that may be applicable to socio-technical systems.

#### 3.1 Neo-Darwinist evolutionary theory

Most biologists subscribe to neo-Darwinist evolutionary theory<sup>1</sup>. This theory integrates the natural selection theories of Darwin with modern genetics (DNA, Mendelian inheritance). Population variance is a product of mutation and recombination of genes. Selection pressures act on these populations, increasing the frequency of positive variations (those that lead to increased reproduction of an individual). As species are isolated, e.g., by geography, new species are created. Fitness is the term used to denote suitability to a particular ecological niche. A niche is comprised of variables such as environment, geography, and other competing species. As the individual changes, so does the environment in which the individuals exists. This is a competitive process.

One way of thinking about fitness is with a ‘fitness landscape’, as shown in Fig 1. On the x-axis is the possible combinations of genes for a given population. The height of the curve along the y-axis denotes replication rate. Genotypes that are higher than others are more fit, and better adapted. Such a model has a clear application in min-max optimizations.

#### 3.2 Other evolutionary proposals

*Complexity theory*, propounded in this context chiefly by Stuart Kauffman (Kauffman, 1995), is a theoretical framework which aims to explain the origin of life using the theory of self-organization. For example, one issue that is unexplained in biology is the emergence of self-replicating molecular chains like RNA and DNA. It isn’t clear how these chains could form from what we understand about the origins of life. Kaufmann’s theory proposes that such complex, organized molecules form because of

---

<sup>1</sup>Discussion in this section adapted from [http://en.wikipedia.org/wiki/Modern\\_evolutionary\\_synthesis](http://en.wikipedia.org/wiki/Modern_evolutionary_synthesis)

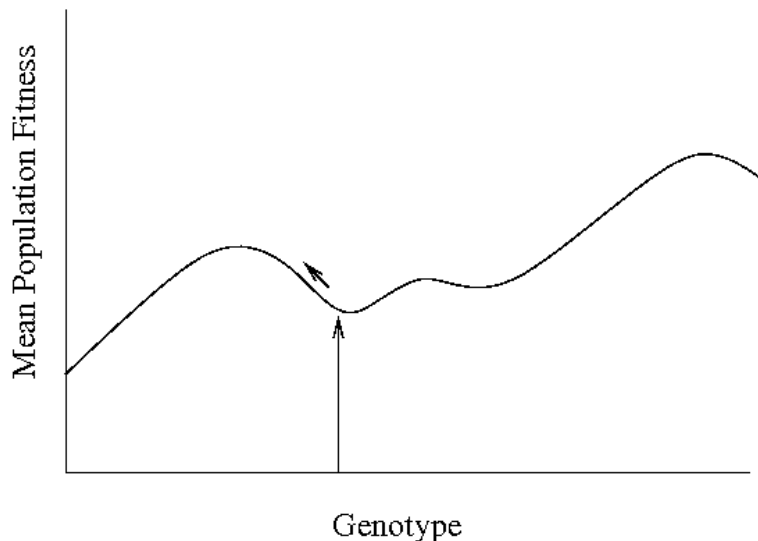


Figure 1: Sample fitness landscape. In this example, a particular genotype mutates positively to find a higher (but not optimal) fitness peak.

emergent properties in chemical reactions. Mapping chemical reaction networks to random graphs, he demonstrates how there is a complexity threshold after which all chemicals would be ‘connected’ with all other chemicals. The details are less important than the notion, often referred to as chaos theory, that there may be stable, well-organized ‘attractors’ in a particular dynamic system, just as there may be other systems where outcomes are unpredictable.

Kaufmann subscribes to neo-Darwinism beyond his ideas on the origin of life, but believes self-organization explains many other phenomena in the world. One of his conceptions is the  $NK$  model, which models the relationship of a particular gene with its co-evolving genes.  $N$  is the number of genes in a genotype, and  $K$  is the number of co-occurring genes in that genotype. This models the influence of other genes on an individual’s fitness. The model can also incorporate a notion of co-operation by incorporating a third dimension,  $C$ , which is the influence of genes from other genotypes. The research on this effect is preliminary.

Further to Kauffman’s model, the concept of *Symbiogenesis* is the notion that two organisms can combine to become one. This is a cooperative, rather than competitive, model of evolution (Margulis and Sagan, 2002). There is other work in the arena of co-operative evolution, such as co-evolution and mutualism. These are models which seek to explain how supposedly competing species come to depend on one another. Game theory, for example the Prisoner’s Dilemma, is one such model.

### 3.3 Applications to socio-technical systems

What are the implications for evolving socio-technical systems, and particular requirements models? First, we should be prepared to abandon the analogy when it fails to help, and avoid overfitting. For example, information systems are maintained by humans, so selection, evolution, and mutation are hardly as random as seen in nature. Secondly, we should recognize that evolution may mean different things. To many it simply means change (over time). However, we could insist on more rigor in this definition: certainly we are mostly interested in *positive* change over time.

Furthermore, to make evolutionary analogies work, we need the three core elements of biological evolution. These are some mutational mechanism, preferably random; a means for ‘reproducing’ and transmitting genetic material; and finally, some fitness criteria to determine which individuals survive. The presence of these elements suggests that a given system can be understood in biological terms.

There are good reasons why evolutionary mechanisms might be useful in the information systems

domain. For example, there are claims in popular literature<sup>2</sup> that software engineering, as currently practiced, cannot develop truly complex systems, such as an analogue to the human mind. The central question is how to produce subsequent generation systems which are more complex (and useful) than the precursors which ‘produced’ them. To some extent, current large-scale system failures argue for an evolved approach. The drawback is that evolutionary techniques are often accompanied by opacity – they don’t explain how the solution was derived. Furthermore, there is a distinction between applying evolutionary techniques to problem-solving, such as generating optimal solutions; and using evolutionary metaphors to better understand the process itself.

Some of the interesting techniques, that are both more and less related to the evolutionary definitions mentioned previously:

Evolutionary computation – These are techniques which are typically based on the neo-Darwinist notions of selection, mutation and reproduction. These are employed to solve problems, typically optimization problems. As stated above, it is necessary to phrase one’s problem in genetic terms, by identifying fitness criteria – which may not be obvious *a priori* – and the DNA of the solution candidates. This is a broad and active research area.

Neural networks – Neural networks are not evolving in the context of this review. Rather, there is phenotypic variation, as weights and inputs are altered. The basic level is essentially unchanged once it is initialized. I don’t believe they would model the evolutionary process.

Probabilistic networks – Much the same as neural nets, probabilistic networks like Bayesian Belief Nets (BBNs) have an essentially unchanging structure once they are initialized. There may be research that examines how to dynamically reconfigure BBN structures, and not just prior and posterior probabilities.

Non-monotonic logics – Discussed in more detail later on, these are logics that provide for belief revision. They can model evolution by allowing the basic unit of information to be a well-formed formula in the logic. Evolution is simulated by adding and removing these *wffs* to the theory. Fitness pressures can be defined that would suggest which formulae ought to be added or removed.

Memetics – Richard Dawkins first postulated the notion of a ‘cultural gene’, which he called a meme. Memes are self-replicating, self-modifying ideas in a culture. A meme might be something like the latest cool slogan. There are some interesting applications to socio-technical systems, because memetics deals with intentional agents, interacting.

Social networks – There has been a great deal of interest lately in the study of social networks, particularly in how these network form and evolve. Kauffman (1995) describes some ways in which these networks might arise without outside interference. The unit of evolution is a node, typically a human actor. Selection pressures can be any outside influence, such as a person’s degree of trustworthiness. Random graphs have typically been used to model such networks, but their predictive power is fairly weak. Another area of social networking is in multi-agent systems. These systems model multiple, possibly distinct agents. Applying evolutionary computing notions like those described above, one could simulate agent interaction and system evolution. Swarm intelligence is one such multi-agent system.

Cellular automata – Popularized in Wolfram (2002), cellular automata are specified as simple algorithms, yet some can produce remarkable complexity. The notion is that such simple algorithms might model natural processes. Such processes might include the spirals on shellfish, or the fractal nature of ferns.

Economics and game theory – There are numerous models which capture motivations and rationale for actors to compete and/or co-operate. Notions like the Nash equilibrium (a satisficing strategy for all players), cooperative auctions, and preference elicitation can all be used to explain patterns in real-world phenomena, including socio-technical systems. Introducing such notions to some of the more traditional requirements models below would be an interesting step.

---

<sup>2</sup>[http://www.technologyreview.com/read\\_article.aspx?id=17089&ch=infotech](http://www.technologyreview.com/read_article.aspx?id=17089&ch=infotech)

One final consideration in running these algorithms are computational complexity constraints. It may be theoretically possible to execute a model simulation, but its running time may not be tractable. For example, the notion that cellular automata might explain all there is in the universe is but one part of the problem. That is, they may indeed explain, but executing them to verify this might take the universe’s lifespan. A good model provides explanatory power while maintaining some level of useful abstraction.

For socio-technical systems research, the primary question surrounds genetic material. What should constitute a allele, gene, and genome in an information system? Su and Mylopoulos (2006) apply Kauffman’s *NKC* model to a model of corporate information systems. They use goals as the base unit of evolution. New goals are coupled via an interface to existing goals, which are in turn related to other, older organizational goals. The model provides a theory for integrating new goals into existing goal models, by explicitly identifying affected/unaffected portions of the original goal model. However, no real use is made of evolutionary modeling, nor is there any sense of how this strategy relates to the notions of fitness and reproduction I have suggested ought to be in evolutionary approaches.

Another approach might be as follows. Again, agent goals are treated as the atomic information elements (i.e., an actor’s goal model is its DNA). We relate goals using notions of dependency and contribution defined later on in this paper. Co-evolution can be explained by monitoring the response of related goals to perturbations in the system (for example, when another goal is marked as satisfied).

We also define an algebra for combining these elements, and for generating new elements. We now need to define a machine or interpreter which can evaluate the result of a given iteration, and possibly produce new instances. Ideally, we would iterate many times over this model to, eventually, produce an optimum arrangement of actors, internal goals, and inter-actor relationships that addresses the fitness criteria (perhaps specified as high-level, system-wide goals).

This example is one way to translate the biological notions of genetics, mutation, selection, and reproduction to socio-technical systems.

## 4 Key concepts in requirements evolution

Having covered some of the background in general notions of evolution, I now turn to evolution in requirements models in particular.

### 4.1 Background terms

To provide a common set of terms for discussing requirements models, I refer to the discussions in Jackson (1997) and Zave and Jackson (1997). They present a fairly straightforward, if somewhat formal, view of requirements and their role in constructing a software-based system. The requirements these papers refer to are functional, in that they can be quantified, measured and translated to operational objects. Neither work characterizes the role of non-functional requirements. This paper treats the two as having equal status in a requirements model, and the issues of evolution apply to both.

The essentials are the presence of an Environment and a Machine (a software-based system). Inputs and outputs of the Machine exist in the Environment. Requirements describe certain phenomena in the Environment (e.g., ‘Elevator A’). Jackson (1997) distinguishes between “those environment properties that are given (indicative descriptions) and those that must be achieved by the machine (optative descriptions)”. Finally, Specifications are sets of relationships over the shared phenomena of the Machine and the Environment. They are a ‘narrow bridge’ between the requirements and the Machine, narrow in that they describe only those properties necessary to build the Machine such that the Requirements are satisfied. The way this relationship is expressed in Jackson (1997, p. 11) is:

$$E, S \vdash R$$

If a machine whose behaviour satisfies  $S$  is installed in the environment, and the environment has the properties described in  $E$ , then the environment will exhibit the properties described in  $R$ .

Implementing a Machine and introducing it hopefully brings about the optative properties, but will also introduce new indicative characteristics in the Environment. Non-functional requirements are meta-properties of the system that cannot be quantitatively related to the specification in  $S$ , but whose status impacts the state of the environment. These include properties like maintainability, usability, and scalability.

A model is an abstraction of the Environment, representing those portions that (we believe) are relevant to the software process. The model is how humans understand the Environment (which cannot be directly perceived). Also, “conceptual models are the artefact closest to the origin of a change, i.e., the end user (Verelst, 2005, p. 469)”.

A good explanation of this process is provided in Zowghi and Gervasi (2003). Figure 2 is taken from this paper. In it, we can see the relationship between specification, requirements, and the Environment (what they call the domain). The node labeled ‘B’ refers to an initial presentation of business requirements. The final result in this model is some Specification. The authors are careful to note that the arrows reflect change. Elements can be changed at each new iteration.

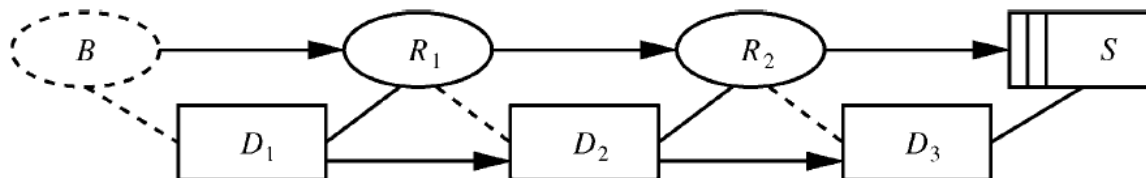


Figure 2: Evolution in Jackson’s model of requirements, specification, and domain. Dashed lines refer to optative requirements, solid lines to indicative requirements. From Zowghi and Gervasi (2003, p. 999).

Evolution in the sense used in this paper then refers to updating the model to reflect changes in the Environment (or domain). The Environment is always changing, due to the uncontrollable and essentially random nature of the world. Those changes can be a result of internal forces – a change brought about by the introduction of the Machine. They can also result from external pressures, that is, changes in the Environment beyond the control of the Machine’s owners. Models can take many forms, from first-order logic, to diagrammatic representations, to natural language descriptions.

For instances of model elements, I say these elements *change*. The truth value of certain predicates (Billy has \$10) changes as time elapses and events occur (Billy gets his paycheque). The model itself, however – that there are elements like Person, Money, and events like Pay-Employee – does not change. In certain frameworks, the distinction between model and instance can be blurred. In this case, any elements can change, regardless of model level. This is the case in Telos (see below). However, practically there remains a distinction between change and evolution. There is a difference between time *in* a model, e.g. studying the progression of a disease; and time *about* a model, e.g. updating a model with new objects, reasoning about the model’s changing characteristics, etc.

Another distinction is between solution uncertainty and problem uncertainty. Harker et al. (1993) define this as the difference between worrying how well the code will address the requirements (the dominant concern), versus how well the requirements address the problem itself. In the terminology of Jackson (1997), this is the same as being overly focused on the bridge to the Machine world, rather than the overall picture of environment and machine (the context).

## 4.2 Evolution taxonomies and frameworks

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 are in the software maintenance community. Rather than present an overarching terminology, I will list pertinent research and draw distinctions between them. I believe the majority of these papers present viable approaches to understanding the concepts involved. Where there are differences, they are the result of perspective.

### 4.2.1 The nature of change

One useful distinction is presented in Mylopoulos (1998). He divides conceptual models into four classes of ontologies (where an ontology is a formal theory of knowledge about a particular domain). These kinds are social, intentional, dynamic, and static models. Time is supported in dynamic models, e.g., processes. These dynamic models manage state and transitions. Other models may address change, but not as explicitly. One can see this separation in the UML family of models: there is a class diagram, as well as an activity diagram describing how those classes interact.

Motivating these system changes are changes in the environment. There are differences in how these requirements arise. Harker et al. (1993) identify these as:

- enduring requirements (core to the business);
- mutable requirements (a product of external pressures);
- emergent requirements (surfaced during thorough elicitation);
- consequential requirements (identified after product implementation – in Jackson’s terms, a new machine affecting the existing environment);
- adaptive requirements (requirements that support system agility); and finally,
- migration requirements (those which help during the changeover).

Where Rajlich and Bennett (2000) and Lientz and Swanson (1980) 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. (1998) 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 which defines them. From generality to extensibility there is an increasing amount of change required for a given requirement; from extensibility to generality there is an 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 (2002, p. 123) (covered in Section 8.1).

Another state space model is covered in Favre (2003), 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 in other words. If we talk about a Machine 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’.

### 4.2.2 Managing change

To understand the motivations behind making changes to a system, a seminal work in the field of software maintenance is Swanson (1976) (see also Lientz and Swanson (1980)). 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 Machine.

Rajlich and Bennett (2000) 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 (2005) for a preliminary assessment).

The process of managing change is also the subject of Nelson et al. (1997). 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/Machine 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 Machine to adapt to a given change event. High levels of the preceding characteristics imply a ready ability to accommodate change. Many proposed requirements engineering frameworks lack change acceptance, relying on users to understand the nature of the change, and manually incorporate it.

Buckley et al. (2005) 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. (2001) 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 enhance. I consider their business rules category to have 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 suggesting fixing requirements problems consist of by far the largest cost in system maintenance (e.g., the Standish reports). 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 an updated requirement. 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 paper refers to.

### 4.2.3 Requirements evolution taxonomies

Many of the prior papers mention requirements only because a presumed change in requirements have driven some software change. However, we are concerned with the nature of these requirements changes. This is also the subject of research by Massimo Felici (Felici, 2003, 2004). In Felici (2003), he refers to requirements evolving in the early phases of a system, with perfective maintenance occurring toward the end of a system’s lifespan. This is at odds with the current view of requirements as something that exists throughout the project lifecycle. More organizational challenges are placed in the taxonomy, as well. Overall, however, the taxonomy was difficult to understand.

In Felici (2004), the analysis is more coherent. He begins with the observation that requirements frameworks generally do a poor job handling evolving requirements. The PROTEUS classification of requirements evolution (Harker et al., 1993) 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 Machine and Environment; adaptive, due to task variation; and migration, arising from planned business changes. I find this taxonomy of causes of requirements evolution to be fairly concise yet comprehensive. Felici also discusses the similar causal taxonomy of Sommerville and Sawyer (1997), which they term ‘volatile requirements’. Sommerville and Sawyer use the categories of mutable, emergent, consequential, and compatibility requirements.

There are naturally other researchers working in what I have termed requirements evolution. However, as they will be the subject of the remainder of this paper, I will refrain from discussion of their terminology until the relevant section.

## 5 Historical perspectives

While I deliberately avoid looking at most aspects of the wider field of software evolution and maintenance, some early history is useful. The early pioneers of the field included Meir Lehman (Lehman, 1986; Lehman and Ramil, 2003) and Fred Brooks (Brooks, 1995). Their research is characterized by empirical analyses of large-scale software systems, particularly IBM’s OS/360 project. The difficulties encountered in this project was the first indication that software ‘engineering’ wasn’t as straightforward as it may have seemed. Lehman proposed several laws of evolution. These were 1) the law of continuing change – that software continues to change, and degenerate, until rebuilt; 2) the law of increasing entropy; 3) the law of statistically smooth growth – despite local variation, all systems follow similar growth patterns. However, his theories emerged from one type of software (he says nothing, for example, about component-based software). As such, they need validation on more robust sets of data. At least one study shows the laws to be false, in particular contexts (Godfrey and Tu, 2000).

Jazayeri (2005) discusses the theoretical nature of software evolution. He maintains that ‘species evolve, individuals age’, his point being that individual software artifacts may not improve (as evolution is commonly understood to be about). There is a real question about what constitutes software in the days of distributed objects, web services, and embedded systems.

In Harker et al. (1993) there is a good description of the importance of understanding requirements evolution. They say, “changing requirements, rather than stable ones, are the norm in systems development (Harker et al., 1993, p. 266).” This is motivated by the understanding that was emerging in the 1990s that the importance of requirements permeated the entire development process, rather than being a strictly ‘up-front’ endeavour.

Examining some of the early models and methodologies is the role of the next section.

### 5.1 SADT

In the early phases of software development, there was little initial awareness of the need for attending to requirements and specification for a software system. Indeed, this attitude prevails today among developers of smaller systems. There is even a pithy acronym: YAGNI – You Ain’t Gonna Need It. However, among developers of larger-scale systems, and after a few failures in earlier days, designers realized that some repeatable process for defining what the system was to do would be essential. For

example, SADT, described in Ross and Schoman (1977), laid out a series of activities and flows in what was termed structured analysis. The goal was to duplicate the repeatable, linear nature of specifications in older disciplines like architecture.

## 5.2 Conceptual Information Modeling

Despite the success of SADT, and related techniques, Bubenko (1980) noted certain flaws. This paper was one of the earliest to note the importance of modeling not just HOW, but also WHAT. It proposed goal modeling as a way to understand the larger system, before drilling down to data modeling or specifications (as in SADT). Bubenko termed his methodology Conceptual Information Modeling, or CIM. It included the notion of a model, and instances thereof, but had no explicit meta-modeling level (the model elements were predefined). The model’s objective was to “explicitly specify abstractions made about the reality and to display all other assumptions made in terms of rules and constraints. (Bubenko, 1980, p. 399)”. Only when this was included with the requirements could a complete, contextual design begin. From an evolution standpoint, his model included two semantics about a requirement: one was the WHAT, that is, the phenomena being modeled, and during which intervals this phenomena holds; the other is the WHEN, the intervals during which the information is required. As we will see, this division presages that of the TELOS system. Note that this treatment of temporal information is at the model level. When we include time in the CIM methodology, we use it to refer to other instances of the model itself. E.g., contracts are valid for some time period. In CIM we also include information about that information, e.g., the information about contract validity is valid as of 2005, until 2007 or otherwise notified.

Interestingly, Bubenko also mentions ‘global system requirements concerning performance, reliability ...’, which we now term non-functional requirements. The paper also notes that, as one of the requirements for a good CIM, “it should make incremental introduction and integration of new requirements easy and natural in the sense that new requirements should require as few changes in an existing model as possible (Bubenko, 1980, p.401).” No guidance is provided as to how to do this, but it does make a clear case for supporting model evolution in any conceptual modeling approach. In essence, the CIM approach is all about continuous refinement of the model, as more information is derived regarding the system and the requirements. To facilitate this, CIM adopts a interval-based event model of time. Events occur at a specific time point and exist indefinitely thenceforth (which seems strange – one is only pregnant for specific intervals).

There are some uncertainties in the description. Although CIM depends on iteration of the model, it isn’t clear how inconsistency is handled. Also not mentioned is how one might specify that a particular predicate is no longer valid (after being updated by another). The mechanism for accomplishing this is not explicit in this paper.

## 5.3 ERAE

The ERAE model (Entity, Relation, Attribute, Event) described in Dubois et al. (1986) is similar to CIM. To motivate the need for the language, the authors emphasize the need for ‘world-oriented requirements’ (system + environment + emergent state) to avoid cost overruns that were at that point well-known in software engineering. They realized knowledge acquisition is difficult in requirements engineering; therefore, this bottleneck should be a priority in an RE language. To solve this, they suggest an object-oriented modeling approach. Part of this includes a vocabulary for expressing object statefulness and events affecting that state (transitions). Associated with the object model is a constraint language expressing static and dynamic constraints. One interesting idea is that events and entities are the same, differentiated only in their lifespan – events are instantaneous. Everything in the model is time-dependent. For example, relationships between entities are possible, not certain. Events are associated with a predicate ‘time’ that specifies when they occur; objects with a predicate ‘exists’ defining lifespan. As an example, consider the ERAE sentence `is-located-at(x,y,t)`. The sentence is true if x is an Elevator, Y is a Floor, and X is at Floor Y at time t. ERAE therefore supports system change by providing a third mandatory property for elements in the system that denotes the time of occurrence or validity. By updating the system (no details are provided on this), one could therefore accommodate

evolving models.

## 5.4 RML

Contemporaneous with ERAE was RML, the Requirements Modeling Language (Greenspan et al., 1994, 1982). RML similarly used a semantic network-like approach of entities and relationships, coupled with logical underpinnings, to construct its model. In both tools there is some use of abstraction (e.g. classification, specialization, decomposition) to enhance the model. RML used time to model history as part of its descriptive approach of a domain. The authors draw a contrast between knowledge representation and conceptual modeling, with conceptual models having more focus on wider scale problems and abstractions over those problems. Recent work has been bridging this apparent gap. The actual approach taken is described as follows:

“The representation of time is essential for languages intended to model dynamic applications, if one is to prevent an implementation bias toward imperative programming style. RML assumes a linear, dense model of time points and encourages history-oriented modeling of an application, which consists of describing possible histories for an entity or activity (or assertion, for that matter). Accordingly, there is a time argument in every predicate appearing in an RML assertion. Moreover time “objects” corresponding to time intervals are constructed as specializations of RML classes (Greenspan et al., 1994, p. 138).”

Again, this is a temporal model that allows one to describe a possible domain state at a given time. Evolving the model consists of updating the various predicates, possibly adding new predicates. However, it isn't clear to what extent updates are supported. CIM, ERAE, and RML all seem oriented to a one-off requirements model that can then be used to design the system (rather than allowing on-the-fly updates and inconsistencies during run-time). In other words, these methodologies assume complete knowledge of the system, e.g., the precise periods for which a concept is applicable.

The subject of this survey is precisely those instances where that information is unknown, such as new policies or situations not known at the initial modeling phase.

## 5.5 Modeling information systems with Telos

The successor to RML was Telos (Mylopoulos et al., 1990). Telos supported formal modeling of four characteristics of information systems: the usage model, the subject model, the system model (the requirements and code), and the development model (coders, processes). It had a formal temporal logic that dealt with time intervals, allowing modelers to specify periods of validity for facts in the model, as well as meta-information regarding when that time period was valid. This model was very flexible. For example, seven temporal relations (and their inverses) were implemented. Every proposition in Telos contains a temporal predicate specifying when that proposition is valid. From a model evolution perspective, this enables history keeping (tracking past changes) and partial validity (over a time interval). Revision in Telos was accomplished with the UNTELL and RETELL operators, but required explicit human intervention.

This paper was also one of the earlier formulations of various organizing hierarchies for information system models. The authors explicitly describe classification (instanceOf), specialization (isA), and aggregation (partOf), as well as others. Telos supported multiple levels of model abstraction. Unlike in CIM, model elements could be constrained with the use of a meta-model specifying entities in the domain. Likewise, this meta-model could be specified with a meta-meta-model, and so on. An example is the chain `letter--documentclass--entity`.

In the ConceptBase product (Jeusfeld et al., 1998), which was a partial implementation of the Telos model, the temporal expressiveness of Telos was eliminated to make reasoning more tractable. They note that Telos presented too much overhead for typical users. This tradeoff is typical of most present-day modeling tools – generally, the ability to reason about time is less important than efficiency in other operations.

## 5.6 The NFR framework

The NFR framework (Chung et al., 1999; Mylopoulos et al., 1992) is a goal-oriented requirements modeling approach that positions non-functional requirements (NFRs) – such as stability, security, and scalability – as the ‘selection criteria’ of a system. This approach focuses on the development *process*, rather than the degree to which the software product meets the NFRs. The framework consists of five elements: goals, which represent objectives and desiderata in the system-to-be; links relating goals, indicating how one goal contributes to another; refinement methods for deriving lower-level goals; correlation rules for determining interactions; and labeling algorithms for assessing design tradeoffs. The aim is to evaluate system design decisions interactively, by assessing the relative satisficing of individual goals. This supports design-time evolution of the system. This helps one choose the initial set of design requirements. To evolve the product following implementation, one must presume that the implemented system follows the existing set of goals, and then update the goal model accordingly (with, for example, new goals reflecting regulatory changes). Automating this process is the focus of current research.

## 5.7 i\*

An extension of the NFR work is presented in Yu (1997). The i\* framework works in the social and intentional areas of conceptual modeling, allowing informal diagrams of organizations, agents, goals, and dependencies. For example, one can make (visual) statements such as “Goal:Schedule Meeting depends on Actor:Meeting Organizer”. This provides a hitherto missing ability to describe the social context a system (Machine) operates in. This approach has been called ‘early-phase RE’. This is a phase that is highly dynamic, during which the model sees constant change as the various stakeholders negotiate what the true nature of an organization is (see the section of Viewpoints, below). As such, there is no explicit temporal support in i\*. Once the i\* model has been derived, it can be operationalized using techniques described in the NFR framework. To evolve an i\* model is not supported, but can be done easily, since i\* products are informal. Often, evolving an i\* product is as simple as redrawing the diagram. More important is the procedural mechanisms for carrying out these changes. That is, who gets to effect a change, who has to be consulted, and so on. Much of this work is ongoing, and described later.

## 5.8 KAOS

KAOS (Dardenne et al., 1993), follows a similar concept to Telos and the NFR framework. Unlike Telos, they also mention the notion of AND/OR hierarchies, in addition to the familiar ones of *isA*, *instanceOf*, and *partOf*. KAOS proposes a goal-decomposition approach to requirements modeling. Evolution, where addressed, is modeled using a variability approach: model all existing knowledge about the system and the environment, and then select, using some satisfiability engine, the alternative that satisfies the various constraints. This is a closed-world assumption: that is, only information known to the modeler is assumed to be true. New information that might invalidate certain assumptions has to be added and modeled anew. KAOS supports temporal operators at the instance level: e.g., property P holds in the current and all future states. This allows one to express temporal constraints. Since it is at the instance level, however, new model-specific information – such as new agents, goals, or constraints – cannot be added or reasoned about (unlike in Telos, as originally formulated).

# 6 Current themes

To provide perspective on current research in requirements evolution, I have identified what I feel are certain major trends. In each section, I discuss relatively recent work (typically since the late 1990s) and place it in context of that theme. To begin this section, I first outline these themes, as well as how they relate.

The themes I have identified are as follows: logical approaches to requirements evolution; viewpoints and inconsistency handling; goals and aspects modeling; requirements management; and agile modeling. Logical approaches assume a Jacksonian view of system building, where certain hard-to-define terms in the Environment are linked to formal concepts in the Machine – the ‘narrow bridge’ of the specification.

Formal in this sense means logically formal, i.e., translatable to a first-order definition. In contrast to this, viewpoints approaches strive to deal with inconsistencies that often arise from conflicting versions of requirements. Evolving requirements often means identifying what functional aspects of a system will be affected. By contrast, a higher level of abstraction characterizes goal modeling approaches to requirements evolution. These efforts often describe the Machine in social and intentional terms, to provide context for rationalizing design choices. Managing requirements is a research area that subsumes others, attempting to provide descriptions of how and when requirements change, in order to provide guidance for future efforts. Traceability is the identification of requirements and linkages between requirements and implementation. Finally, agile approaches are fairly recent alternatives to more traditional, structured requirements techniques. As such, they provide a nice bookend to the opening discussion of formal requirements models.

## 6.1 Formal approaches

These approaches rely on languages with well-defined model theories, such as first-order logic and extensions thereof. The advantage to these approaches is that they provide a concise and complete definition of the state of the world, and they provide a set of sentences for which consistent inferences can be produced. The chief disadvantage is that most requirements problems do not lend themselves to formalization. Typically this is because the requirements are poorly defined or misunderstood. In some cases, as will be addressed in the following section, the requirements are inconsistent and unformalizable some formal inconsistency handling language is used. The other disadvantages are: 1) formal approaches sometimes lack the tractability to permit reasoning over large models or theories; 2) formal approaches are often difficult for non-logicians to understand, making adoption a challenge.

There are many languages that have been proposed for dealing with change, revision, and statefulness. Some of these include temporal logics, Kripke structures, state machines, and various forms of situation calculus. Acknowledging this, I consider only logical approaches to requirements evolution.

In requirements evolution, the typical approach has been to first model the existing set of requirements formally, then specify formal transitions from the initial model  $M$  to a new model,  $M'$ , that satisfies the conditions that caused the evolution. Most effort has gone into defeasible reasoning such as belief revision (Ghose, 1999; Rodrigues et al., 2004; Zowghi and Offen, 1997). Other work has been in Markov models (Whittaker and Poore, 1993) and abductive reasoning and machine learning (Garcez et al., 2001). Finally, a formal language admits but one interpretation, and is therefore useful for explicitly stating system properties in analysis problems (Aaltonen and Mikkonen, 2002; Mens and Eden, 2005).

### 6.1.1 Defeasible requirements models

Detecting and handling inconsistency is a strength of formal requirements methods, since formal language require (at some point) consistent theories. In order to generate consistent theories, defeasible reasoning is a popular approach. Defeasible reasoning implies that stated axioms are tentatively held. A popular approach is belief revision; axioms are valid until new knowledge invalidates pre-existing conclusions. At that point, some way of (partially) ordering the axioms is needed to determine what axioms to reject.

Zowghi and Offen (1997) use a logic based on defaults (known as AGM) to make certain initial assumptions about the Machine to be. Then, as new information arises, these defaults are revised. A partial order on the axioms is proposed, called *epistemic entrenchment*, which allows more deeply held beliefs to be retained in each iteration of the formal specification. These are essentially weightings applied to the various axioms. This belief revision approach is fairly powerful, assuming one can derive the initial formal model, and that the model is not overly large.

In the Zowghi/Offen model, each new formal model is consistent, although inconsistent with previous versions. Ghose (1999) considers this an overly strong requirement, and suggests a modification that permits *non-prioritized* model evolution. Non-prioritized change means that new axioms are added to the model, then compared with existing information to see whether the update should be permitted. He defines sets of essential (may not be changed) and tentative requirements (which may be changed). To evolve a specification, a choice function can be defined over the models which chooses one of the possible candidates that will resolve inconsistencies in the next iteration. Ghose's approach is advantageous in

that it permits the operator to define choice functions over the models according to what requirements should remain. Although he claims this is preferred to the non-explicit approach in Zowghi and Offen (1997), it would seem fairly similar to their notion of epistemic entrenchment.

Finally, Rodrigues et al. (2004) use a model they call *clustered belief revision* to accomplish much the same result. In that model, requirements can be clustered into sets, and a partial order defined over those sets. This partial order defines which formulation of the theory will be chosen at a particular iteration of the model. Their approach centres around evolving a specification prior to implementation, which isn't true model evolution as I use the term. Nonetheless, the technique could readily be applied in that context.

### 6.1.2 Specifying system properties

Another use of formal language is to delineate the properties of the system. Machine learning techniques take these formal models, typically specifications, and use statistics and simulation to learn the most probable results. For example, Whittaker and Poore (1993) uses statistical techniques, known as Markov chains, to model software change. They use probabilistic predictions of software usage with a (ideally formal) software specification. This produces a set of independent states of software usage that the given specification permits. Once this is derived, probabilistic transitions are derived for each state. These can be uniformly assigned, or distributed based on observational data. The model is then 'run', to produce further refinement of the probability distributions. This produces some idea of how this idealized version of the specification would behave, e.g., how many times a fault would occur.

Two challenges seem apparent. One is deriving all possible states a piece of software might be in. This is actively explored in formal methods research. The second challenge is what to make of the data. The authors use the concept to explore 'Clean-room' software engineering, which targets defect prevention. What use this statistical model might have for evolving the software isn't explained. However, it would seem that one could alter the Markov chain to introduce new situations, in order to see how well the specification could adapt.

Garcez et al. (2001) relies on the existence of a sound theory and a set of observations about a system. Abductive reasoning is applied to derive some possible explanations for the set of observations. Abductive reasoning, which is reasoning from a theory and implications to derive antecedent conditions, or explanations (in contrast to inductive and deductive reasoning). Inductive reasoning in the form of a backpropagating neural network then derives potential consistent evolved models from that set of explanations. There could be other explanations, but the neural network, assuming relevant training data, should produce the most likely explanation. In this way a (logically consistent) revised specification can be produced. The challenge in this approach, aside from scalability, is deriving minimally useful set of training data.

### 6.1.3 Explicit language

The final use of logical formalism I cover is in making analysis explicit. Logical implication is less important than a rigorous syntax for discussing system properties. Mens and Eden (2005), for instance, use logic to describe something called *evolution complexity*. They define this as determining the degree of effort required to evolve a particular system. Applying this concept to design patterns such as Visitor, they assess the *shifts* in requirements and the implications for the resulting implementation. The advantage of formalism here is a concise explanation of the implications of a particular design change. This allows for high-level reasoning about the complexity of a given change, similar to computational complexity notions like Big O notation.

Aaltonen and Mikkonen (2002) use formal language to describe how changes to high-level system abstractions – much like design patterns – will impact lower-level implementation. They term this an *abstraction hierarchy*. The term is used in a different context in cognitive work analysis (Section 8.1). The model is state-based, and refinement of an abstraction level results in a progressively more operationalized state transition diagram. Changed requirements can be modeled in terms of an existing abstraction hierarchy, and the level of abstraction provides some understanding of the cost associated with adopting that requirement. The formal model is used to concisely describe the implication of implementing these requirements.

Formal Tropos Fuxman et al. (2001) is a formalization of the  $i^*$  language designed to marry the use of early requirements languages with the power of model-checking tools. It incorporates a temporal semantics into  $i^*$  to allow for the expression of assertions about dependencies in the system; e.g., prior-to task A, task B must be fulfilled.

Summing up formal approaches to requirements evolution is the paper by Ghose (1999). Four criteria are presented for requirements evolution frameworks: 1) minimal change during revisions; 2) trade-off analysis of each revision; 3) prevent premature commitment; and 4) support reuse.

## 6.2 Semi-formal inconsistency handling in requirements evolution

Inconsistency in requirements is not the same as evolving a requirements model. During the elicitation phase, a model may have inconsistencies, which are resolved during the first implementation. However, the newly consistent model is now subject to the vagaries of the changing Environment. Most of the work presented in this section, while focused on deriving a workable first implementation, could also be adapted to evolving a requirements model for various versions of an implementation.

Inconsistency handling is desirable in requirements engineering because inconsistency, at least initially, is a common situation. Among stakeholders, for example, there may be different views on what the current context is, and what a new system ought to look like. Some of the formal models mentioned previously, such as Ghose (1999), can also be considered inconsistency management models. They temporarily allow for models which conflict with one another, then provide a formal process for resolving that conflict.

The papers discussed in this section, however, are either semi-formal or informal. The viewpoints framework, and related notions of mediated inconsistency, have strong parallels with research in argumentation and design rationale. For a survey, refer to Shum and Hammond (1994). Argumentation approaches take an informal approach based on dialectic, rather than logic.

The Viewpoints framework, covered in Easterbrook and Nuseibeh (1995), is a method for maintaining inconsistent and separate requirements models of the same Machine. Three advantages are proposed for this approach: stakeholder appeal, separation of modeling tasks, and delayed commitment to the final model. Current research suggests some of these advantages are not meaningful in large projects, however (Easterbrook et al., 2005). A Viewpoint consists of some modeling notation, alongside a history of changes to the model, and some rules for resolving inconsistencies. Each stakeholder can model his/her requirements and understanding of the domain, and resolve inconsistencies as he/she sees fit – ignore or explore the inconsistency. At the end of the modeling, inconsistencies are resolved and the specification can be derived. However, there appears to be no reason why this approach could not be applied across implementations of a particular Machine. Each Viewpoint then represents a temporal actor. New requirements can be assessed in light of the existing viewpoint (possibly the integrated model from the initial modeling phase). Inconsistencies suggest potential regression test failures, and point out areas of maintenance concern.

Scenarios, in the most simple sense, describe a Machine and the tasks associated with that Machine. In Breitman (2000); Breitman et al. (2005) the authors consider scenarios requirements models that are defined in the language of the user. They propose a more complete integration of scenarios with the software development process. As such, support is needed for evolving these scenarios. There are two dimensions: new scenarios which emerge as the Machine evolves, and old scenarios that must be similarly updated. Breitman and Leite propose a framework to manage the relations between these scenarios. Each scenario is represented as a series of explicit steps. Then, the relationships between these scenarios is captured according to a set of relations such as subset, exception, and precedence. When the Machine evolves, the model can provide insight into which scenarios require updates.

Viewpoints can be used to represent different perspectives on scenarios, as in Hui and Ohnishi (2003). They use scenarios as sequential process descriptions, and represent them formally. They show, with a simple example, that integrating the scenarios first, then evolving the joint scenario, is faster than evolving viewpoints on that scenario, then integrating the result. Integration is done by mapping scenario components using a set of relationships.

Finally, one can also use simulation to manage inconsistency. Seybold et al. (2004) use notions from software testing to permit semi-formal testing of requirements models. In their case, they use a

representation similar to statecharts, and propose adding operators that specify incompleteness of the model, e.g., abstract relationships. Stubs are generated that show the model element exists, but that its behavior is undefined. In this way, the model can be iteratively constructed with partial information at each stage. Then, semi-automatic modeling techniques can be used to assist in requirements elicitation. This approach can also be used throughout a product’s lifecycle. New requirements can be added as stubs, then the model ‘tested’ to determine how the new model would perform. Such a testing framework is clearly dependent on the choice of modeling formalism. Statecharts seem to lend themselves well to this notion.

### 6.3 Goals, features, non-functional requirements, and aspects

This section groups a variety of separate research areas in requirements engineering. The common theme among them is a focus on higher-level and non-functional aspects of a Machine, that are often represented by an explicit model.

#### 6.3.1 Features

For example, we might talk about a collection of features in a particular product (which may themselves be functions such as Spellcheck). With new versions of software, the features (observable behaviour) available have evolved. Anton and Potts (2001) describe such an evolution process in the context of telephony. Features may be used to extract the requirements model where no good information exists. For example, Hsi et al. (2003) and Anton and Potts (2001) both try to model an existing Machine by better understanding the features offered by that system – its outputs to the Environment, in other words. Hsi and Potts (2000) takes a similar approach, producing indicative descriptions of the system in order to inform optative requirements for new features. They describe the system’s morphology (its appearance to the user), the system functions, and an object model comprised of the previous two. Feature modeling is a large research area that is largely beyond the scope of this paper.

#### 6.3.2 Aspects

Aspects are a set of cross-cutting system concerns – which may be functional – that aren’t captured by the existing system modeling hierarchy (e.g., decomposition, classification). By bringing aspects to the fore, we hope to better understand the hidden complexities of a Machine. For example, security code may be woven into many separate objects in an object-oriented design. To simplify maintenance, and potentially evolution, aspect-oriented approaches collect the relevant security code in one place, and then use special syntax to re-integrate the code to the appropriate place.

Tourwé et al. (2003) argue that aspect-oriented development may complicate the software evolution process, since languages such as AspectJ introduce new artifacts into the source-code, increasing coupling. However, the paper does not substantiate this claim.

With more empirical support, Coady and Kiczales (2003) argue that aspects can greatly simplify evolving systems. They looked at evolution of the BSD operating system over three versions. Compared with traditional software development, they argue that aspects allowed for maintenance of cross-cutting code in one location, as well as providing explicit configuration indications. They note that aspects might evolve independently. The requirements they deal with are functional, but it is easy to see how non-functional concerns could be added at a higher level of abstraction.

Araujo and Ribeiro (2005) use aspects slightly differently. They suggest that scenario-based requirements models might identify cross-scenario commonalities (such as fault situations). These commonalities might be collected in a scenario aspect, then woven into the scenarios as needed. If a new scenario emerged, the existing aspects might well remain. For example, an ATM that permitted a new operation on an account would still have situations where faults occurred, e.g. wrong card, wrong PIN, etc.

Aspects can be identified through requirements analysis. Yu and Mylopoulos (2004) proposes a framework for extracting aspects from goal decomposition. An algorithm generates a candidate list of aspects, such as data integrity, using a goal structure called a V-graph. Nuseibeh (2004) similarly proposes better linking of requirements modeling with concern identification, but notes that requirements may prove less consistent than might be desirable.

The explicit identification proposed in aspect-oriented modeling helps with developing a rational evolutionary strategy. Centralizing requirements, or portions of requirements, can help reduce the complexity of future updates.

### 6.3.3 Goals and NFRs

Goals and non-functional requirements (NFRs) were introduced in Section 5.6. Goals are a way of structuring a Machine according to intentionality. We can have a Machine with deliberate desires, seeking to satisfy those desires. In intentional models like  $i^*$ , we also have other participants, like users, managers, etc., with their own sets of desires. Non-functional goals are a way of specifying requirements, or system desires, which cannot be quantified. We say these goals are satisficeable, that is, we can be content with the level of fulfilment, but we cannot agree that it is ever completely satisfied. This concept is based on the work of Simon (1967).

Since most goal modeling techniques take a fairly high-level view of Requirements, they can be well-suited to managing evolution. Many of the relevant frameworks were discussed earlier, including the NFR framework, Telos,  $i^*$ , and KAOS. The most common approach to handling evolution is the support of implementation variability (discussed in the following section).

Chung et al. (1996) use the NFR framework to model evolution of a banking system. Rather than incorporating the potential change in the initial model, they evolve the model by adding new goals and decompositions. The advantage is that any new goals can be accommodated and understood within the NFR framework; however, the process is entirely manual.

The GBRAM framework presented in Antón (1996) explicitly considers goal evolution:

...[G]oals are characteristically more stable than the processes, organizational structures and operations of a system which continuously evolve; this is why we emphasize them. Nevertheless, goals evolve gradually and informally depending on the changing needs, circumstances and goal priorities of stakeholders (Antón, 1996, p. 137).

However, this is in the context of one-time system development, as they note that evolution only occurs until the specification is complete. Evolution in this instance is a way to elaborate and refine requirements during system creation.

Ghose (1999) argues that his requirements evolution framework sits before approaches like NFR. It attempts to reconcile non-functional requirements (**response time < 2 min**) and functional requirements (**compute position**). It describes some logical techniques for resolving these conflicts. This model seems to ignore concerns raised by research in complex, component-based systems (see Section 6.4.1 for more).

Goal models are also mentioned in Rolland et al. (2004) and Etien and Salinesi (2005). In considering organizational pressures on requirements, the authors describe their goal-oriented *map* approach. Maps are expressions of intentions and strategies. Gaps describe differences between maps of the indicative and optative states of the system. They contribute a useful typology of potential gaps in two maps (differences in goal models, in other words). The combination of the indicative map and a statement of optative properties, expressed in the gap typology as a pattern of evolution, produces an evolved map representing the Machine-to-be. They argue that this takes less time to construct than a brand-new Machine would.

### 6.3.4 Variability

Research is looking at ways to automate, at least partially, the process of translating high-level goals into specifications and design. The idea is to allow users to manipulate goal graphs directly, with the changes reflected in lower-level details. Evolving the model would result in an evolved software product (either implementation or specification). The central question is whether this model evolution is planned for, or added after the fact. Most variability approaches assume the former. This may be presumptuous, if the change is unexpected or unanticipated.

Goal models are one way of managing this, as described in Letier and van Lamsweerde (2004); Mylopoulos et al. (2001), among others. During the construction of the goal model, alternatives are

specified, typically by using some form of AND/OR decomposition. Liaskos et al. (2006) calls these ‘AND-facets’. Different versions can be specified by following different branches through the goal tree. To evolve a system entails predicting future changes, modeling those as goals, and switching to that particular variation as needed. For example, a database server may not initially need to scale to more than 100 requests per second. However, the designers may estimate a need for scalability of 10,000 requests per second. This would be modeled as an alternative in the goal model. Goal models have been proposed as useful ways to enable different configurations in self-managed, autonomic systems (Lapouchnian et al., 2005).

Product lines – software products sharing commonalities, but differing in specific areas, are another approach, as in Buhne et al. (2005). There is also a role for probabilistic approaches that attempt to predict which variation will be required (Letier and van Lamsweerde, 2004).

Finally, there is close intersection with research in configurable workflows, as described in Section 8.2.

## 6.4 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. (1998) 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 (1981), 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. (2004) 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 (1999) 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. (2004) discuss one approach for managing architectural evolution in sync with code. Another approach is to ignore everything but the source code, and reverse engineer requirements from there Yu et al. (2005).

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. (2005) will be important here.

### 6.4.1 COTS and components

Another key issue is the use of different components in a software project. These components may not be under one’s direct control. Such components are called Commercial, Off-The-Shelf components, or COTS. A change in one component, driven by an evolution in a particular requirement, might impact other components. Etien and Salinesi (2005) 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 about up to 10 months between new releases, and generally unsupported by their vendors after three subsequent releases.” (Boehm, 2006, p. 9)

The work that led to that analysis, Yang et al. (2005), 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 (2001) 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.

#### 6.4.2 Traceability

Traceability is an aspect of requirements management that links a phenomenon in the Environment to some phenomenon in the Machine. Traceability is essential to 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 (1994). One needs a trace from the various Environmental phenomena to the specification of the requirements for the Machine. 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, covered in Fickas and Feather (1995), and extended in Feather et al. (1998), is one mechanism for tracing between requirements and code. Monitoring involves inserting code into a Machine to determine how well requirements are being met. A monitor records the usage patterns of the system, such as numbers of licences in use. This information can be extracted and used to evolve the Machine, 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. Challenges in control systems approaches to constructing a Machine are also covered in Section 8.1. Monitoring also has influence in various research into autonomic systems, which are self-managing systems (Kephart and Chess, 2003).

Traceability is more difficult in non-functional requirements, because by definition these requirements do not have quantitative satisfaction criteria. Cleland-Huang et al. (2005) discusses a probabilistic information retrieval mechanism for recovering non-functional requirements from class diagrams.

Ramesh and Jarke (2001) give a lengthy overview of empirical studies of requirements traceability.

### 6.5 Agile modeling

Agile modeling is derived from the notion of agile software development. Advocates of that approach argue for:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan<sup>3</sup>

In this context, agile modeling is light-weight modeling along the principles of using the minimal amount of infrastructure required, while maximizing stakeholder values Ambler and Jeffries (2002). Many agile software methods argue that modeling is unnecessary, but in large software projects, some form of model is usually desirable. More research is required on the intersection of modeling and agility, e.g., is there a lower limit on ‘model-ness’? What are the benefits and limitations to an agile approach?

---

<sup>3</sup>from <http://agilemanifesto.org>

One study (Botaschanjan et al., 2004) looked at using requirements models as testing objects in an agile development process. They define requirements in UML, model constraints in the Object Constraint Language, and then verify the requirements by generating code and running test cases. They claim this can provide valuable early feedback (similar to the approach in Seybold et al. (2004)).

Integrating requirements throughout the development lifecycle – as promoted in Nuseibeh (2001) and Boehm (2006), is similar to this philosophy. Both promote iterative analysis and early identification of requirements and architectural demands. One of the principles of agile software techniques is to avoid ‘Big Design Up-Front’. Agile modeling recognizes that change is inevitable, i.e., one of the principles is that it is impossible to identify all requirements in the beginning. Change is to be accommodated and managed, not ignored.

Agile methods might be a good way of surfacing requirements (Schwaber, 2002). One can consider this a ‘constant-prototyping’ approach; Paetsch et al. (2003) consider agile development ‘adaptive rather than predictive’. In other words, versions of the Machine are in almost continuous contact with the Environment. This generates new understanding about how the Machine/Environment interaction functions; this in turn develops understanding about adaptive requirements, perfective requirements, and so on. A counter-argument – that hasn’t been studied – is that an agile process will by definition have trouble with cross-cutting concerns, such as a scalability. Some of these topics are mentioned in Eberlein et al. (2002); Lindvall et al. (2002); Paetsch et al. (2003); Schaefer and Kutschera (2002). Most mention non-functional requirements as ‘system drivers’, determining the focus for future development, but there is no guidance on how these cross-functional concerns are handled in an agile method, which typically focuses on small teams and rapid iteration. From Paetsch et al. (2003): “Agile methods need to include more explicitly the handling of nonfunctional requirements.”

## 7 Empirical studies of evolution in requirements models

Some empirical work has been described throughout this review, for example, in requirements management. There is some empirical work in the broader software maintenance community; Kemerer (1995) provides an overview. Early works looked at, for example, whether procedural or object-oriented models were easier to maintain.

One can approach the issue of evolution by seeking to make the initial conceptual model high quality, e.g., anticipating future changes. Genero et al. (2000) presents metrics to assess ER diagram quality, and presents a controlled experiment to demonstrate this. A later study (Genero et al., 2004) looks at quality metrics in UML diagrams, showing which metrics suggest greater maintainability costs.

This is somewhat unsatisfying, however. The research effort seems directed at anticipating evolvability and managing change in the initial (pre-release) model. True, there is a big gain here, as often this is the only time requirements models are consulted. But, as Bennett and Rajlich (2000) note, “the fundamental problem, supported by 40 years of hard experience, is that many changes actually required are those that the original designers cannot even conceive of.”

Is it unrealistic to think that comprehensive, initial models will be revisited? Evidence seems to suggest that change occurs regardless: Verelst (2005) mentions a study (Marche, 1993) which demonstrated that companies with outdated ER models would change the definitions in the system rather than the model itself, suggesting that either the model is brittle and/or that it is expensive to change the model.

One can also question to what extent the level of abstraction of a model influences maintainability and evolvability of the model. Verelst (2005) studies whether model abstraction impacts the ease of evolving a model. Many studies and evangelists mention abstraction as a key method of handling system adaptation (e.g., design patterns); however, abstraction is also seen as making comprehension more difficult. Verelst found that for numerous simple changes, an abstract (UML) model was indeed preferable; however, novices struggled to make changes to the abstract model without degrading the overall structure. He concludes that abstractions may be of less use than previously assumed (although the experiment used small models and novice modelers). There is a great deal of research on model comprehension that is beyond the scope of this review.

There has been some attempt to understand the process and products of requirements evolution, as in Anderson and Felici (2000, 2001, 2002) and Felici (2003). This work examined two industrial case

studies. A key contribution is the development of a Requirements Maturity Index (RMI), a measure of how frequently requirements are changed. The authors note that certain requirements, such as high-level architectural concerns, are fairly stable, while other requirements are less so. A flaw in the RMI is that not all requirements changes are equal in effort or ‘pain’, as established in Basili and Weiss (1981). Quantitative approaches in general must be sensitive to the fact that many requirements changes are social in nature, a product of business needs, socio-organizational style, and other factors.

A few studies have examined requirements volatility, largely in the software maintenance context. Stark et al. (1999) collected data about requirements change in seven products and generated descriptive statistics about the process. However, they were restricted to one organization. It seems clear that organizational processes have a major role in driving requirements change. Consequently, generalizability in these studies is challenging. To address this, Ferreira et al. (2003) propose the use of simulation models to understand causal factors. Like all simulations, its accuracy depends on the initial training data. Case studies (Lindvall and Sandahl, 1998; Loconsole and Borstler, 2005; Lutz, 2003; Nanda and Madhavji, 2002; Nurmuliani et al., 2004) are used to generate descriptive models of change processes. Like the previous two studies, the intention was to better comprehend the complexities behind requirements volatility. A better understanding of this process is essential in correctly supporting requirements model evolution.

It is clearly an important issue. For example, the implications for validation and verification downstream from requirements changes are the focus of an industrial/academic collaboration described in Heumesser et al. (2003). This was a project examining evolution in the context of real-time systems. Surveys of large industrial partners give some idea of the interest these players have in understanding requirements evolution.

One question in empirical evolution research concerns replicability. Demeyer et al. (2001) propose a framework for evaluating software for potential benchmarks in research. They conclude that few tools support the analysis/design phase of software development, i.e. few tools have readily assessed requirements or design artifacts. This makes evaluation difficult. For instance, one popular tool for research is the Mozilla web browser. However, there are no good documents identifying requirements for this tool. The requirements that exist are in the form of email records and newsgroup postings. A serious qualitative research program would be required to surface these requirements.

## 8 Other forms of requirements evolution

There are other notions of evolution in related areas of software-based systems. Examining some of these other areas might provide a useful cross-section of evolution concepts.

### 8.1 Complex systems models

I differentiate these from previous models only in terms of the type of systems under investigation. Many of the preceding examples used simple systems as the basis for the proposed Machine. These systems include meeting schedulers, for example, or other, typically closed-world systems. On the other hand, many problems fall into the category of complex sociotechnical systems. These systems – such as industrial plants or semi-autonomous software in healthcare – are usually studied in systems and industrial engineering. I argue that this distinction grows somewhat meaningless as software pervades all systems, and as ‘traditional’ software engineering projects grow more complex.

There are several approaches to modeling such systems. I describe the Cognitive Work Analysis (Vicente, 1999) perspective as I am most familiar with it.

CWA is a five stage analysis model going from structural work domain constraints to the cognitive constraints of the operator. There is no explicit notion of evolution in this model. That is, the analysis is completed and the system is designed accordingly. Vicente (1999) claims that the best application of CWA is in ‘revolutionary’ designs, that is, where there are no pre-existing system constraints. This is a formative approach to analysis – design of new systems. This is perhaps due to historical applications to relatively stable industries.

This is no longer so clearly the case. For instance, the CWA perspective, and its cousin Ecological Interface Design, have been applied to analysis of complex database systems (Duez and Vicente, 2005).

	Anticipated	Unanticipated
Familiar	<i>Standard procedure</i>	<i>Unapproved procedure</i>
Unfamiliar	<i>Emergency procedure</i>	<i>No procedure</i>

Table 1: Event typology of Vicente and Rasmussen (1992). Designers fill the X-axis of anticipation of potential occurrences. Operators/users occupy the Y-axis. Faults occur in the unanticipated and unexpected areas of the matrix.

In that work, the need for some way of managing changing system properties, perhaps through the use of default reasoning, was noted.

An important contribution of CWA is regarding the language of system analysis. By using a fairly strict terminology, CWA research is fairly precise about what is being analyzed. There has been extensive empirical grounding for the approach. The nature of change in systems is one such contribution. Vicente and Rasmussen (1992) noted three types of change events, expanded on by Jamieson (2006). These are illustrated in Table 1. The contribution of this typology is to establish the range of events which can occur, and to emphasize that even the best design will exhibit events in all cells of this matrix. Consequently, the CWA literature argues for a formative design that permits the operator to understand the constraints imposed by the system and its Environment, and act accordingly.

There are gradations to how likely designers feel specific events may be – such as a 100-year flood event. Under the guise of risk management, these events may be assigned a certain probability, and if they fall beneath a threshold, then nothing will be done to ameliorate them. The cost of anticipating them is greater than the expected utility of doing so. Interesting work in risk management has been presented in Kiper and Feather (2005).

Once a catalog of these events is generated, it is clearly desirable to update operating procedures and system design to remove the source of unanticipated and unfamiliar events, or to mitigate them. It is here where model evolution can play a role. Identifying the elements of the design – however expressed – is crucial to updating the Machine.

On the other hand, complex systems models, such as CWA, offer experience with complexity that many requirements engineering frameworks are only beginning to be applied to. Understanding change and evolution in CWA, then, offers good lessons for requirements engineering.

## 8.2 Flow models

Business process modeling is an increasingly important research area. The primary difference from software requirements is the explicit focus on flows – of information or of tasks. A workflow, for example, is a model of task and information relationships in some organization. There are also business process models, and rule languages, such as the Business Process Execution Language (BPEL), which can be used to compose workflows and execute them.

Most workflows aren't static, as noted in Reichert and Dadam (1998), although most languages assume this is the case. Their research proposes a formal theory for modeling workflows. It provides for a set of change operations users can invoke, and correctness and consistency checking of these proposed changes (along the lines of the discussion in 6.1). Along the same lines, Ellis and Keddara (2000) describe a workflow system based on colored Petri nets that can respond (formally) to dynamic situations.

What are these changes? Kumar and Narasipuram (2006) discuss a taxonomy of levels of anticipated change in workflow models. Many of these notions closely parallel those discussed in complex systems engineering. For example, both share the idea that abnormal events are often unanticipated by designers, and often there is no clear process to follow to correct the abnormality. Kumar and Narasipuram (2006) use a four stage model, moving from constant stimulus, through uncertain and ambiguous stimulus, to surprise stimulus.

The main interest for the requirements researcher is naturally the focus on execution and process modeling. This is an aspect of the Machine that many of the previous works have omitted in favour of structural descriptions of the Environment.

### 8.3 Database schemas

Database schema are conceptual models of database information structures. There is a rich field of research concerning how best to evolve these schema as information requirements change. Database schemas can be simpler than information and organizational models. There has been a great amount of work done in the database world on temporal databases, that is, database systems which can handle temporal relations. Here, we should distinguish between the majority of these efforts, which seek to capture data validity (when the data was valid in the real world), called valid-time. The remainder, which align with the scope of this paper, represent the validity of the schema itself (e.g. the model of the world, and not the individuals in that model). This area is typically identified as schema evolution. The key questions concern the mapping of an existing schema to a new schema, that reflects certain changes in the world.

Roddick (1995) surveys issues in schema versioning, and presents evolution as a subset of the more general versioning problem. Evolution is schema modification, and versioning is retrospective and prospective accessing of data in a schema (which necessitates understanding the evolution of the schema). Miller et al. (1993) discusses the notion, in the database community, of schema equivalence. Schemas are considered equivalent if one can update the same data in either version. Roddick comments that this condition is very strong, and implies backwards-compatibility for all pre-existing schemas; easier is to support viewing of data according to prior schemas, but updating only through the current schema.

The reason this field has relevance in requirements evolution is the relatively large amount of research into various issues that arise in evolving an information model. For example, there is work on temporal data models and temporal query languages. These results should prove helpful in the requirements domain.

### 8.4 Ontology evolution

The final area I wish to touch on is ontology evolution. Ontologies are formalized conceptualizations of a domain of interest, and therefore closely map to requirements models. Interesting ontologies are nearly always formally grounded. Since they attempt to describe domains, they naturally evolve as understanding of the domain (e.g., medicine) changes. Recent ontologies built for use on the Web suffer from the COTS-software problem: a desire to integrate with other, third-party ontologies. Noy and Klein (2004) explain that with web-based ontology languages such as OWL, a key design goal is to support distributed ontologies and ontology reuse. As such, one might have applications that were designed by organizations entirely outside the ontology provider, who rely on a specific ontology structure to function.

Noy and Klein (2004) make several important points regarding the distinctions between database schemas and ontologies. 1) ontologies are data and model combined; 2) ontologies have semantics (whereas databases have only relational semantics), which causes changes beyond what a human might do; 3) ontologies see more reuse; 4) ontologies are linked and decentralized; 5) ontologies are richer in representation primitives; 6) ontology classes and instances are often the same (via metaclasses). Many of these differences apply also to requirements models and database schemas.

Research in ontology evolution is emerging. Quite a bit of research has been done on integrating ontologies, or versioning different ontologies, e.g. Noy and Musen (2002). Easterbrook (2003) uses arguments from requirements engineering to argue that inconsistency might be acceptable in semantic web applications. This suggests that these two areas might be quite similar.

## 9 Research opportunities

The next milestone in my Ph.D. program is the research proposal. This is the preliminary proposal for research that will lead to several studies and ultimately, a full-fledged research proposal. My research is at the intersection of traditional ‘process’ models – workflow, sequencing, business processes – and ‘product’ models – object diagrams, strategic rationale, etc. My experience with the CWA framework has taught me the importance of an initial separation of these discrete models, followed by the use of each to inform the other.

Large-scale software systems increasingly are built with a loosely-coupled design paradigm. For example, service-orientation in the myGrid project (Stevens et al., 2004) permits a degree of customization unknown before that design strategy. However, loose coupling has disadvantages, and one such problem lies in evaluating the (possibly independent) service: who runs it? Can I trust its results (e.g., sequence data, disease classification)? Furthermore, what should be done when one component evolves? It could define new operations, remove existing ones, or alter the result of previous services? How does one manage this? Jazayeri (2005) discusses the difference between software aging and design evolution. How does this notion describe the aging of a service, with the evolution of a design? If one component changes, does the entire workflow change? These are all important questions.

To answer some of these questions, I outline the following preliminary research proposal. I propose a five-part structure for my dissertation. These elements would be:

1. **A theory of requirements evolution**, incorporating the notions of constraints into requirements models, as well as some of the notions from Section 3. Many of the methodologies in use do a poor job of managing evolution and changing requirements. Design tools are heavily formative and prescriptive, rather than reactive and agile. There are two potential contributions in this area. One is to apply notions from approaches such as CWA to requirements analysis tools (or vice versa). The other interesting angle is applying organizationally derived policies (an explicit specification of intent) to complex models, and work on aligning those models with high-level policy-based requirements. See, for example, Madhavji and Tasse (2003).
2. **A goal algebra** for querying/altering requirements models. This algebra would be based on similar notions in the tool *grok* (Wu et al., 2002); the language would be used to manipulate the structure of goal models. Such manipulations would achieve two things: one, address scalability issues with current tools; two, provide for experimentation with evolving goal models.
3. **A visualization framework** describing what is important to see, perhaps as in Blundell and Pettifer (2004). The framework would outline the requirements for visualizing – and comprehending – requirements models, in particular evolving models. It would then be applied to existing tools to assess them. This is similar to work I did in Ernst et al. (2005).
4. **A tool for storing requirements models** and configurations, possibly like the Molhado system of Nguyen et al. (2005). Current requirements tools predominantly store requirements in databases. Work on traceability links those artifacts with design and implementation artifacts, but there is no explicit way to handle changing requirements, particularly post-implementation. Such a tool would need to be used in conjunction with process improvements.
5. **Sample data/case studies** of evolving models. Such data would serve to validate the approaches outlined in the four previous components. Such a dataset should have some meaningful amount of changes in its history. Defining meaningful would be part of this research. Some ideas for datasets include extracting model histories from source – either software source code, or perhaps organizational and legal policies. In Wu and Holt (2006), the authors describe a repository of source code changes in open-source software. There may be a way of extracting requirements models from that corpus. A final option may be to partner with an industrial organization such as IBM, perhaps mining a tool like Rational Software Architect.

## References

- T. Aaltonen and T. Mikkonen. Managing software evolution with a formalized abstraction hierarchy. In *Eighth IEEE International Conference on Engineering of Complex Computer Systems*, pages 224–231, 2002.
- S. W. Ambler and R. Jeffries. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. Wiley, Boston, March 2002.
- S. Anderson and M. Felici. Controlling requirements evolution: An avionics case study. In F. Koornneef and M. van der Meulen, editors, *Intl Conf on Computer Safety, Reliability and Security*, volume LNCS 1943, pages 361+, Rotterdam, 2000.
- S. Anderson and M. Felici. Requirements evolution: From process to product oriented management. In *International Conference on Product Focused Software Profess Improvement*, LNCS 2188, pages 27–41, Kaiserslautern, Germany, September 2001. Springer.
- S. Anderson and M. Felici. Quantitative aspects of requirements evolution. In *Intl. Computer Software and Applications Conference*, pages 27–32, 2002.
- A. I. Antón. Goal-based requirements analysis. In *Intl. Conf. on Requirements Engineering*, pages 136–144, Colorado Springs, Colorado, April 1996. IEEE Computer Society. Most influential paper award at RE06.
- A. I. Anton and C. Potts. Functional paleontology: system evolution as the user sees it. In *International Conference on Software Engineering*, pages 421–430, Toronto, Canada, 2001.
- J. Araujo and J. C. Ribeiro. Towards an aspect-oriented agile requirements approach. In *International Workshop on Principles of Software Evolution*, pages 140–143, 2005.
- V. R. Basili and D. M. Weiss. Evaluation of a software requirements document by analysis of change data. In *Intl Conf. on Software Engineering*, pages 314–323, San Diego, USA, 1981. IEEE Press.
- K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: a roadmap. In *Conference on The Future of Software Engineering at ICSE*, pages 73–87, Limerick, Ireland, June 2000. ACM Press.
- B. Blundell and S. Pettifer. Requirements for the visualisation of ontological evolution. In *Theory and Practice of Computer Graphics*, pages 18–23, Bournemouth, UK, June 2004.
- B. Boehm. Some future trends and implications for systems and software engineering processes. *Systems Engineering*, 9(1):1–19, 2006.
- B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.
- J. Botaschanjan, M. Pister, and B. Rumpe. Testing agile requirements models. *Journal of Zhejiang University SCIENCE*, 5(5):587–593, 2004.
- K. Breitman. Scenario evolution: a closer view on relationships. In *International Conference on Requirements Engineering*, pages 95–105, 2000.
- K. K. Breitman, J. C. S. do Prado Leite, and D. M. Berry. Supporting scenario evolution. *Requirements Engineering*, 10(2):112–131, 2005.
- F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, Boston, August 1995.
- J. Bubenko. Information modeling in the context of system development. In *IFIP Congress*, pages 395–411, 1980.

- J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, September 2005.
- S. Buhne, K. Lauenroth, and K. Pohl. Modelling requirements variability across product lines. In *13th IEEE International Conference on Requirements Engineering*, pages 41–50, 2005.
- N. Chapin, J. E. Hale, J. F., Ramil, and W.-G. Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, 2001.
- L. Chung, B. A. Nixon, and E. Yu. Dealing with change: An approach using non-functional requirements. *Requirements Engineering*, 1(4):238–260, December 1996.
- L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*, volume 5 of *International Series in Software Engineering*. Springer, October 1999.
- J. Cleland-Huang, R. Settimi, O. Benkhadra, E. Berezanskaya, and S. Christina. Goal-centric traceability for managing non-functional requirements. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 362–371, New York, NY, USA, 2005. ACM Press.
- Y. Coady and G. Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In *AOISD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 50–59, New York, NY, USA, 2003. ACM Press.
- A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, April 1993.
- S. Demeyer, T. Mens, and M. Wermelinger. Towards a software evolution benchmark. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 174–177, New York, NY, USA, 2001. ACM Press.
- E. Dubois, J. Hagelstein, E. Lahou, F. Ponsaert, and A. Rifaut. A knowledge representation language for requirements engineering. *Proceedings of the IEEE*, 74(10):1431–1444, 1986.
- P. Duez and K. J. Vicente. Ecological interface design and computer network management: The effects of network size and fault frequency. *International Journal of Human-Computer Studies*, 63(6):565–586, December 2005.
- S. Easterbrook. Semantic integration and inconsistency. In A. Doan, A. Halevy, and N. Noy, editors, *Semantic Integration Workshop (at ISWC)*, Sanibel Island, Florida, October 2003.
- S. Easterbrook and B. Nuseibeh. Managing inconsistencies in an evolving specification. In *Second IEEE International Symposium on Requirements Engineering*, pages 48–55, 1995.
- S. Easterbrook, E. Yu, J. Aranda, Y. Fan, J. Horkoff, M. Leica, and R. A. Qadir. Do viewpoints lead to better conceptual models? an exploratory case study. In *International Conference on Requirements Engineering*, pages 199–208, 2005.
- A. Eberlein, S. Greenspan, and J. Leite, editors. *International Workshop on Time-Constrained Requirements Engineering*, Essen, Germany, September 2002.
- C. Ellis and K. Keddara. Ml-dews: Modeling language to support dynamic evolution within workflow systems. *Comput. Supported Coop. Work*, 9(3-4):293–333, 2000.
- N. A. Ernst, M.-A. Storey, and P. Allen. Cognitive support for ontology modeling. *International Journal of Human-Computer Studies*, 62(5):553–577, May 2005.
- A. Etien and C. Salinesi. Managing requirements in a co-evolution context. In *13th IEEE International Conference on Requirements Engineering*, pages 125–134, Paris, September 2005.

- J.-M. Favre. Meta-model and model co-evolution within the 3d software space. In *Intl. Wshp on Evolution of Large-scale Industrial Software Applications at ICSM*, Amsterdam, September 2003.
- M. S. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behaviour. In *Ninth IEEE International Workshop on Software Specification and Design (IWSSD-9)*, pages 50–59, Isobe, JP, April 1998.
- M. Felici. Taxonomy of evolution and dependability. In *Proceedings of the Second International Workshop on Unanticipated Software Evolution, USE 2003*, pages 95–104, Warsaw, Poland, April 2003.
- M. Felici. *Observational Models of Requirements Evolution*. PhD thesis, University of Edinburgh, 2004. URL <http://homepages.inf.ed.ac.uk/mfelici/doc/IP040037.pdf>.
- S. Ferreira, J. Collofello, D. Shunk, G. Mackulak, and P. Wolfe. Utilization of process modeling and simulation in understanding the effects of requirements volatility in software development. In *Workshop on Process Software Simulation Modeling (PROSIM) at ICSE*, May 2003.
- S. Fickas and M. S. Feather. Requirements monitoring in dynamic environments. In *RE '95: Proceedings of the Second IEEE International Symposium on Requirements Engineering*, Washington, DC, USA, 1995. IEEE Computer Society.
- A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model checking early requirements specifications in tropos. In *International Symposium on Requirements Engineering*, pages 174–181, Toronto, August 2001.
- D. A. S. Garcez, A. Russo, B. Nuseibeh, and J. Kramer. An analysis-revision cycle to evolve requirements specifications. In *Intl. Conference on Automated Software Engineering*, San Diego, USA, November 2001.
- M. Genero, L. Jimnez, and M. Piattini. Measuring the quality of entity relationship diagrams. In A. H. F. Laender, S. W. Liddle, and V. C. Storey, editors, *ER 2000: 19th International Conference on Conceptual Modeling*, volume LNCS 1920, pages 513–526, Salt Lake City, Utah, USA, October 2000. Springer Berlin / Heidelberg.
- M. Genero, M. Piattini, and E. Manso. Finding "early" indicators of uml class diagrams understandability and modifiability. In *Intl. Symp. Empirical Software Engineering*, pages 207–216, 2004.
- A. K. Ghose. A formal basis for consistency, evolution and rationale management in requirements engineering. In *International Conference on Tools with Artificial Intelligence*, pages 77–84, 1999.
- M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, pages 131–142, Washington, DC, USA, October 2000. IEEE Computer Society.
- O. C. Z. Gotel and C. W. Finkelstein. An analysis of the requirements traceability problem. In *International Conference on Requirements Engineering*, pages 94–101, 1994.
- S. Greenspan, J. Mylopoulos, and A. Borgida. On formal requirements modeling languages: Rml revisited. In *International conference on Software engineering*, pages 135–147, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
- S. J. Greenspan, J. Mylopoulos, and A. Borgida. Capturing more world knowledge in the requirements specification. In *International conference on Software engineering*, pages 225–234, Tokyo, 1982. IEEE Computer Society Press.
- S. D. P. Harker, K. D. Eason, and J. E. Dobson. The change and evolution of requirements as a challenge to the practice of software engineering. In *IEEE International Symposium on Requirements Engineering*, pages 266–272, 1993.

- N. Heumesser, A. Trendowicz, D. Kerkow, H.-G. Gro, and L. Loomans. Essentials and requisites for the management of evolution requirements and incremental validation. Technical report, Evolution Management and Process for EMPRESS: Evolution Management and Process for Real-Time Embedded Software Systems, July 2003. URL [http://www.empress-itea.org/deliverables/D1.3\\_v1.0\\_Public\\_Version.pdf](http://www.empress-itea.org/deliverables/D1.3_v1.0_Public_Version.pdf).
- I. Hsi and C. Potts. Studying the evolution and enhancement of software features. In *Proceedings of the 2000 International Conference of Software Maintenance*, pages 143–151+, San Jose, CA, October 2000. IEEE Press.
- I. Hsi, C. Potts, and M. Moore. Ontological Excavation: Unearthing the core concepts of the application. In *10th Working Conference on Reverse Engineering*, pages 345–352, Victoria BC, 2003. IEEE.
- Z. H. Hui and A. Ohnishi. Integration and evolution method of scenarios from different viewpoints. In *International Workshop on Principles of Software Evolution*, pages 183–188, 2003.
- M. Jackson. The meaning of requirements. *Annals of Software Engineering*, 3:5–21, 1997.
- G. A. Jamieson. Ecological interface design for petrochemical process control: An empirical assessment. *IEEE Transactions on Systems, Man and Cybernetics (A)*, 36(5), 2006.
- M. Jazayeri. Species evolve, individuals age. In *International Workshop on Principles of Software Evolution*, pages 3–9, 2005.
- M. A. Jeusfeld, M. Jarke, H. W. Nissen, and M. Staudt. Conceptbase - managing conceptual models about information systems.
- S. A. Kauffman. *At Home in the Universe: The Search for the Laws of Self-Organization and Complexity*. Oxford University Press, Oxford, UK, 1995.
- C. Kemerer. Software complexity and software maintenance: A survey of empirical research. *Annals of Software Engineering*, 1(1):1–22, December 1995.
- J. Kephart and D. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1), Jan 2003.
- J. D. Kiper and M. S. Feather. A risk-based approach to strategic decision-making for software development. In *Hawaii International Conference on System Sciences*, page 313. IEEE Computer Society, 2005.
- K. Kumar and M. M. Narasipuram. Defining requirements for business process flexibility. In *Workshop on Business Process Modeling, Development, and Support (BPMDS'06) Requirements for flexibility and the ways to achieve it (CAiSE'06)*, Luxembourg, June 2006.
- A. Lapouchnian, S. Liaskos, J. Mylopoulos, and Y. Yu. Towards requirements-driven autonomic systems design. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, July 2005.
- M. M. Lehman. Modes of evolution. In M. Dowson, editor, *Int Software Process Workshop*, pages 29–32, Breckenridge, Colorado, 1986. IEEE Computer Society.
- M. M. Lehman and J. F. Ramil. Software evolution - background, theory, practice. *Inf. Process. Lett.*, 88(1-2):33–44, 2003.
- E. Letier and A. van Lamsweerde. Reasoning about partial goal satisfaction for requirements and design engineering. In *Symposium on Foundations of Software Engineering*, pages 53–62, Newport Beach, CA, November 2004. ACM Press.
- S. Liaskos, A. Lapouchnian, Y. Yu, E. Yu, and J. Mylopoulos. On goal-based variability acquisition and analysis. In *Intl. Conf. Req. Engineering*, September 2006.
- B. P. Lientz and B. E. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, 1980.

- M. Lindvall and K. Sandahl. How well do experienced software developers predict software change? *J. Syst. Softw.*, 43(1):19–27, October 1998.
- M. Lindvall, V. Basili, B. Boehm, P. Costa, K. Dangle, F. Shull, R. Tesoriero, L. Williams, and M. Zelkowitz. Empirical findings in agile methods. In D. Wells and L. Williams, editors, *Second XP Universe and First Agile Universe Conference*, volume LNCS 2418, pages 197–207, Chicago, IL, USA, August 2002.
- A. Loconsole and J. Borstler. An industrial case study on requirements volatility measures. In *Asia-Pacific Software Engineering Conference*, pages 1–8, 2005.
- M. Lormans, H. van Dijk, A. van Deursen, E. Nocker, and A. de Zeeuw. Managing evolving requirements in an outsourcing context: an industrial experience report. In *International Workshop on Principles of Software Evolution*, pages 149–158, 2004.
- R. R. Lutz. Operational anomalies as a cause of safety-critical requirements evolution. *Journal of Systems and Software*, 65(2):155–161, February 2003.
- N. H. Madhavji and J. Tasse. Policy-guided software evolution. In *International Conference on Software Maintenance*, pages 75–82, Amsterdam, September 2003.
- S. Marche. Measuring the stability of data models. *European Journal of Information Systems*, 2(1):37–47, 1993.
- L. Margulis and D. Sagan. *Acquiring Genomes: A Theory of the Origins of Species*. HarperCollins, Toronto, January 2002.
- T. Mens and A. H. Eden. On the evolution complexity of design patterns. *Electronic Notes in Theoretical Computer Science*, 127(3):147–163, April 2005.
- R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. The use of information capacity in schema integration and translation. In *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases*, pages 120–133, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.
- J. Mylopoulos. Information modeling in the time of the revolution. *Information Systems*, 23(3-4):127–155, 1998.
- J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing knowledge about information systems. *Information Systems*, 8(4):325–362, 1990.
- J. Mylopoulos, L. Chung, and B. Nixon. Representing and using non-functional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, 18(6):483–497, 1992.
- J. Mylopoulos, L. Chung, S. Liao, H. Wang, and E. Yu. Exploring alternatives during requirements analysis. *IEEE Software*, 18(1):92–96, 2001.
- V. Nanda and N. H. Madhavji. The impact of environmental evolution on requirements changes. In *International Conference on Software Maintenance*, pages 452–461, Montreal, October 2002.
- K. M. Nelson, H. J. Nelson, and M. Ghods. Technology flexibility: conceptualization, validation, and measurement. In *International Conference on System Sciences*, volume 3, pages 76–87, Hawaii, 1997.
- T. N. Nguyen, E. V. Munson, J. T. Boyland, and C. Thao. An infrastructure for development of object-oriented, multi-level configuration management services. In *International Conference on Software Engineering*, pages 215–224, St. Louis, MI, May 2005. ACM Press.
- N. Niu, S. Easterbrook, and M. Sabetzadeh. A category-theoretic approach to syntactic software merging. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, Budapest, Hungary, September 2005. IEEE Computer Society.

- N. F. Noy and M. Klein. Ontology evolution: Not the same as schema evolution. *Knowledge and Information Systems*, 6(4):428–440, July 2004.
- N. F. Noy and M. A. Musen. Promptdiff: A fixed-point algorithm for comparing ontology versions. In *National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence*, pages 744–750, Edmonton, Alberta, Canada, July 2002.
- N. Nurmuliani, D. Zowghi, and S. Powell. Analysis of requirements volatility during software development life cycle. In *Australian Software Engineering Conference*, pages 28–37, 2004.
- B. Nuseibeh. Weaving together requirements and architectures. *IEEE Computer*, 34(3):115–119, 2001.
- B. Nuseibeh. Crosscutting requirements. In *International conference on Aspect-oriented software development*, pages 3–4, Lancaster, UK, 2004. ACM Press.
- F. Paetsch, A. Eberlein, and F. Maurer. Requirements engineering and agile software development. In *IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 308–313, Linz, Austria, 2003.
- V. T. Rajlich and K. H. Bennett. A staged model for the software life cycle. *IEEE Computer*, 33(7):66–71, 2000.
- B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93, 2001.
- M. Reichert and P. Dadam. Adept<sub>flex</sub>-supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems*, 10(2):93–129, March 1998.
- J. F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, 1995.
- O. Rodrigues, A. Garcez, and A. Russo. Reasoning about requirements evolution using clustered belief revision.
- C. Rolland, C. Salinesi, and A. Etien. Eliciting gaps in requirements change. *Requir. Eng.*, 9(1):1–15, 2004.
- R. Roshandel, A. V. D. Hoek, M. Mikic-Rakic, and N. Medvidovic. Mae – a system model and environment for managing architectural evolution. *ACM Trans. Softw. Eng. Methodol.*, 13(2):240–276, April 2004.
- D. T. Ross and K. Schoman. Structured analysis for requirements definition. *Transactions on Software Engineering*, 3(1):6–15, 1977.
- D. Rowe, J. Leaney, and D. Lowe. Defining systems evolvability - a taxonomy of change. In *International Conference and Workshop: Engineering of Computer-Based Systems*, pages 45+, Maale Hachamisha, Israel, April 1998. IEEE Computer Society.
- S. Schaefer and P. Kutschera. Applying agile methods in rapidly changing environments. Technical report, IBM Unternehmensberatung GmbH, Munich, DE, July 2002. URL <http://www.jeckstein.com/papers/AgileMethods-SteffenSchaefer&#38;PeterKutschera.pdf>.
- K. Schwaber. The impact of agile processes on requirements engineering. In *Proceedings of the International Workshop on Time Constrained Requirements Engineering*, pages 12–16, September 2002.
- C. Seybold, S. Meier, and M. Glinz. Evolution of requirements models by simulation. In *7th International Workshop on Principles of Software Evolution*, pages 43–48, 2004.
- S. B. Shum and N. Hammond. Argumentation-based design rationale: what use at what cost? *International Journal of Human-Computer Studies*, 40(4):603–652, April 1994.

- H. A. Simon. Motivational and emotional controls of cognition. *Psychological Review*, 74(1):29–39, January 1967.
- I. Sommerville and P. Sawyer. *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, New York, NY, USA, April 1997.
- G. Stark, A. Skillicorn, and R. Ameele. An examination of the effects of requirements changes on software releases. *Crosstalk: Journal of Defence Software Engineering*, pages 11–16, December 1998.
- G. E. Stark, P. Oman, A. Skillicorn, and A. Ameele. An examination of the effects of requirements changes on software maintenance releases. *Journal of Software Maintenance: Research and Practice*, 11(5):293–309, 1999.
- R. Stevens, R. Mcentire, C. Goble, M. Greenwood, J. Zhao, A. Wipat, and P. Li. mygrid and the drug discovery process. *Drug Discovery Today: BIOSILICO*, 2(4):140–148, July 2004.
- N. Su and J. Mylopoulos. Evolving organizational information systems with tropos. In *Conference on Advanced Information Systems Engineering*, 2006.
- H. Svensson and M. Host. Introducing an agile process in a software maintenance and evolution organization. In *European Conference on Software Maintenance and Reengineering*, pages 256–264, Manchester, UK, March 2005.
- B. E. Swanson. The dimensions of maintenance. In *Intl. Conf. on Software Engineering*, pages 492–497, San Francisco, California, 1976. IEEE Computer Society.
- T. Tourwé, J. Brichau, and K. Gybels. On the existence of the aosd-evolution paradox. In *AOSD 2003 Workshop on Software-engineering Properties of Languages*, Boston, USA, 2003.
- J. Verelst. The influence of the level of abstraction on the evolvability of conceptual models of information systems. *Empirical Software Engineering*, 10(4):467–494, October 2005.
- K. Vicente and J. Rasmussen. Ecological interface design: theoretical foundations. *Systems, Man and Cybernetics, IEEE Transactions on*, 22(4):589–606, 1992.
- K. J. Vicente. Ecological interface design: Progress and challenges. *Human Factors*, 44:62–78, 2002.
- K. J. Vicente. *Cognitive Work Analysis: Toward Safe, Productive, and Healthy Computer-Based Work*. Lawrence Erlbaum Associates, New Jersey, April 1999.
- J. A. Whittaker and J. H. Poore. Markov analysis of software specifications. *ACM Trans. Softw. Eng. Methodol.*, 2(1):93–106, January 1993.
- K. E. Wiegers. Automating requirements management. *Software Development Magazine*, 7(7), July 1999.
- S. Wolfram. *A New Kind of Science*. Wolfram Media, January 2002.
- J. Wu and R. C. Holt. Seeking empirical evidence for self-organized criticality in open source software evolution. Research report CS-2006-14, School of Computer Science, University of Waterloo, April 2006. URL <http://swag.uwaterloo.ca/~j25wu/papers/CS-2006-14.pdf>.
- J. Wu, A. E. Hassan, and R. C. Holt. Using graph patterns to extract scenarios. In *International Workshop on Program Comprehension*, pages 239–247, Paris, France, June 2002.
- Y. Yang, J. Bhuta, B. Boehm, and D. N. Port. Value-based processes for cots-based applications. *IEEE Software*, 22(4):54–62, 2005.
- E. S. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *International Symposium on Requirements Engineering*, pages 226–235, Annapolis, Maryland, 1997.

- Y. Yu and J. Mylopoulos. From goals to aspects: discovering aspects from requirements goal models. In *International Requirements Engineering Conference*, pages 33–42, Kyoto, Japan, September 2004.
- Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, and A. Lapouchnian. Reverse engineering goal models from legacy code. In *International Conference on Requirements Engineering*, pages 363–372, 2005.
- P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, 1997.
- D. Zowghi and V. Gervasi. On the interplay between consistency, completeness, and correctness in requirements evolution. *Information and Software Technology*, 45(14):993–1009, November 2003.
- D. Zowghi and R. Offen. A logical framework for modeling and reasoning about the evolution of requirements. In *Third IEEE International Symposium on Requirements Engineering*, pages 247–257, 1997.