Self-Adaptive Software: Landscape and Research Challenges

Mazeiar Salehie and Ladan Tahvildari
Software Technologies Applied Research (STAR) Group
University of Waterloo, Waterloo, Canada
{msalehie, ltahvild}@uwaterloo.ca

Software systems dealing with distributed applications in changing environments normally require human supervision to continue operation in all conditions. These (re-)configuring, troubleshooting, and in general maintenance tasks lead to costly and time-consuming procedures during the operating phase. These problems are primarily due to the open-loop structure often followed in software development. Therefore, there is a high demand for management complexity reduction, management automation, robustness, and achieving all of the desired quality requirements within a reasonable cost and time range during operation. Self-adaptive software is a response to these demands; it is a closed-loop system with a feedback loop aiming to adjust itself to changes during its operation. These changes may stem from the software system's *self* (internal causes e.g., failure) or *context* (external events e.g., increasing requests from users). Such a system is required to *monitor* itself and its context, *detect* significant changes, *decide* how to react, and *act* to execute such decisions. These processes depend on adaptation properties (called self-* properties), domain characteristics (context information or models), and preferences of stakeholders. Noting these requirements, it is widely believed that new models and frameworks are needed to design self-adaptive software. This survey article presents a taxonomy, based on concerns of adaptation, i.e., *how*, *what*, *when* and *where*, towards providing a unified view of this emerging area. Moreover, as adaptive systems are encountered in many disciplines, it is imperative to learn from the theories and models developed in these other areas. This survey article presents a landscape of research in self-adaptive software by highlighting relevant disciplines and some prominent research projects. This landscape helps to identify the underlying research gaps and elaborates on the corresponding challenges.

Categories and Subject Descriptors: A.1 [**General Literature**]: Introductory and Survey; D.2.9 [**Software**]: Management—*Software configuration management*; *Software quality assurance*; C.4 [**Computer Systems Organization**]: Performance of Systems; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software Maintenance*

General Terms: Management, Performance, Reliability, Design

Additional Key Words and Phrases: Self-Adaptive Software, Survey, Self-* properties, Taxonomy

1. INTRODUCTION

Scientists and engineers have made significant efforts to design and develop self-adaptive systems. These systems address adaptivity in various concerns including performance, security, and fault management [Laddaga 1999; Kephart and Chess 2003]. While self-adaptive systems are used in a number of different areas, this article focuses only on their application in the software domain, called *self-adaptive software*. Researchers in this area have proposed several solutions to incorporate adaptation mechanisms into software systems. In this way, a software application which would normally be implemented as an open-loop system, is converted to a closed-loop system using feedback. While adaptivity may be achieved through

feed-forward mechanisms as well (e.g., through workload monitoring), the feedback loop takes into account a more holistic view of what happens inside the application and its environment.

Self-adaptive software aims to adjust various artifacts or attributes in response to changes in the *self* and in the *context* of a software system. By *self*, we mean the whole body of the software, mostly implemented in several layers, while the *context* encompasses everything in the operating environment that affects the system's properties and its behavior. Therefore, in our view, self-adaptive software is a closed-loop system with feedback from the *self* and the *context*.

A fundamental question is why we need self-adaptive software. The primary reason is the increasing cost of handling the complexity of software systems to achieve their goals [Laddaga 2000]. Among these goals, some deal with management complexity, robustness in handling unexpected conditions (e.g., failure), changing priorities and policies governing the goals, and changing conditions (e.g., in the context of mobility). Traditionally, a significant part of the research on handling complexity and achieving quality goals has been focused on software development and its internal quality attributes (as in ISO 9126-1 quality model [ISO/IEC 9126-1 2001]). However, in recent years, there has been an increasing demand to deal with these issues at operation time (run-time). The primary causes for this trend include an increase in the heterogeneity level of software components, more frequent changes in the context/goals/requirements during run-time, and higher security needs. In fact, some of these causes are consequences of the higher demand for ubiquitous, pervasive, embedded, and mobile applications, mostly in the Internet and ad-hoc networks.

Self-adaptive software is expected to fulfill its requirements at run-time in response to changes. To achieve this goal, software should have certain characteristics, known as *self-* properties* [Kephart and Chess 2003; Babaoglu et al. 2005]. These properties provide some degree of variability, and consequently, help to overcome deviations from expected goals (e.g., reliability). Managing software at run-time is often costly and time-consuming. Therefore, an adaptation mechanism is expected to trace software changes and take appropriate actions at a reasonable cost and in a timely manner. This objective can be achieved through monitoring the software system (*self*) and its environment (*context*) to detect changes, make appropriate decisions, and act accordingly. Required changes in traditional software systems can stem from different categories of maintenance/evolution, as discussed in a relevant IEEE standard [IEEE-ISO/IEC 14764 2006]. This standard discusses corrective maintenance for fixing bugs, adaptive maintenance for adjusting the software according to changing environments, perfective maintenance for updating the software according to changing requirements, and finally, preventive maintenance for improving software maintainability. Although this standard does not explicitly refer to dynamic/run-time changes (dynamic evolution) in conjunction with these four categories, these changes are part of what is needed to deal with bugs and new/changing requirements. Dynamic/runtime changes are the basis for adaptation in self-adaptive software.

This survey article presents an overview of basic principles, properties, and background behind self-adaptive software. It proposes a taxonomy of adaptation relying

on the questions of *when*, *what*, *how*, and *where*. Using this taxonomy, a landscape is presented by reviewing a number of disciplines dealing with self-adaptive software, as well as some selected research projects. The comparison between different views followed in this landscape provides a framework for identifying challenges, current research gaps, and some promising future directions.

The rest of this article is organized as follows: Section 2 discusses the principles of self-adaptive software by reviewing the underlying definitions, properties, requirements, and processes. Section 3 introduces a taxonomy for self-adaptive software by relying on a hierarchical structure capturing the adaptation requirements. Section 4 explains how the techniques developed in the context of adaptive systems in several existing disciplines can be used to support self-adaptive software, and then analyzes some selected projects to identify research gaps. Section 5 aims to identify future research challenges and classifies them according to the specified research gaps. Finally, Section 6 presents a summary and some concluding remarks.

2. SELF-ADAPTIVE SOFTWARE: PRINCIPLES AND REQUIREMENTS

This section presents a general review of the basic concepts in self-adaptive software. The objective is to provide a unified set of definitions, goals, and requirements that are used in the rest of the article.

2.1 Definition

Among several existing definitions for self-adaptive software, one is provided in a DARPA Broad Agency Announcement (BAA) [Laddaga 1997]: "Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible." A similar definition is given in [Oreizy et al. 1999]: "Self-adaptive software modifies its own behavior in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation."

Prior to formalizing the concept of self-adaptive software, there has been a related point of view regarding the adaptive programming principle as an extension of object-oriented programming [Lieberherr and Palsberg 1993]: "A program should be designed so that the representation of an object can be changed within certain constraints without affecting the program at all." According to this view point, an adaptive program is considered as: "A generic process model parameterized by graph constraints which define compatible structural models (customizers) as parameters of the process model." This view on adaptation is similar to reflection and meta-programming techniques.

In another point of view, adaptation is mapped to evolution. Buckley *et al.* provide a taxonomy of evolution based on the object of change (*where*), system properties (*what*), temporal properties (*when*), and change support (*how*) [Buckley et al. 2005]. Salehie *et al.* map this taxonomy to self-adaptive software domain, and propose a conceptual model for adaptation changes based on Activity Theory [Salehie et al. 2009]. Static and dynamic adaptation, related to the temporal dimension of this view, are mapped to compile-time evolution and load-time/run-time evolution, respectively. For this reason, dynamic adaptation is sometimes called *dynamic evo-*

lution. In fact, self-adaptivity is linked to what Lehman has discussed on feedback and feedback control in the context of the software process for evolution [Lehman 1996]. According to this article, the essence of self-adaptive software is aligned with the laws of evolution (described by Lehman and his colleagues in FEAST/1 and FEAST/2 projects).

Self-adaptive software systems are strongly related to other types of systems. The notable ones are autonomic and self-managing systems [Kephart and Chess 2003]. However, it is difficult to draw a distinction between these terminologies. Many researchers use the terms self-adaptive (not specifically self-adaptive software), autonomic computing, and self-managing interchangeably, for instance in the survey provided by Huebscher and McCann [Huebscher and McCann 2008]. In comparing self-adaptive software to autonomic computing, there are some similarities and some differences. From one point of view, the self-adaptive software domain is more limited, while autonomic computing has emerged in a broader context. This means self-adaptive software has less coverage and falls under the umbrella of autonomic computing. From another point of view, we can consider a layered model for a software-intensive system that consists of: application(s), middleware, network, operating system, hardware [McKinley et al. 2004], and sub-layers of middleware [Schmidt 2002]. According to this view, self-adaptive software primarily covers the application and the middleware layers, while its coverage fades in the layers below middleware. On the other hand, autonomic computing covers lower layers too, down to even the network and operating system (e.g., see reincarnation server in Minix 3.0 [Tanenbaum and Woodhull 2006]). However, the concepts of these domains are strongly related and in many cases can be used interchangeably.

The key point in self-adaptive software is that its life-cycle should not be stopped after its development and initial set up. This cycle should be continued in an appropriate form after installation in order to evaluate the system and respond to changes at all time. Such a closed-loop deals with different changes in user requirements, configuration, security, and a number of other issues.

2.2 Self-* Properties

Adaptivity properties are often known as self-* properties. One of the initial well-known set of self-* properties, introduced by IBM, include eight properties [IBM-AC 2001]. This section discusses these properties, along with some other related ones, towards providing a unified hierarchical set, which will be used in the rest of the article.

2.2.1 *A Hierarchical View.*

Figure 1 illustrates a hierarchy of self-* properties in three levels. In this hierarchy, self-adaptiveness and self-organizing are general properties, which are decomposed into major and primitive properties at two different levels. The rest of this section further elaborates on each level of this hierarchy.

—**General Level**: This level contains global properties of self-adaptive software. A subset of these properties, which falls under the umbrella of self-adaptiveness [Oreizy et al. 1999], consists of *self-managing*, *self-governing*, *self-maintenance* [Kephart and Chess 2003], *self-control* [Kokar et al. 1999], and *self-evaluating* [Laddaga 2006]. Another subset at this level is *self-organizing* [Jelasity et al. 2006; Seru-

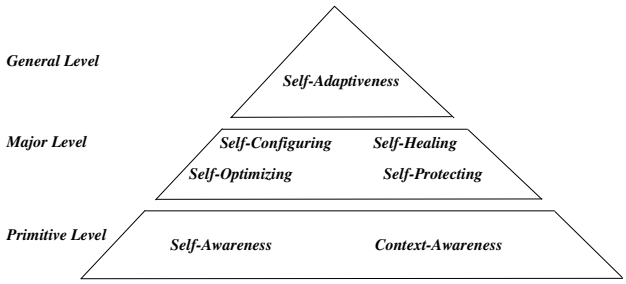

Fig. 1: Hierarchy of the Self-* Properties

gendo et al. 2003], which emphasizes decentralization and emergent functionality(ies). A system with such a property typically consists of many interacting elements that are either absolutely unaware of or have only partial knowledge about the global system. The self-organizing property is bottom-up, in contrast to self-adaptiveness, which is typically top-down. Although most of the concepts in this article are applicable to the self-organizing property, this property is not the primary concern of this survey work. Noting the amount of research dealing with self-organizing systems, a separate survey would be needed to adequately cover this emerging area.

—**Major Level**: The IBM autonomic computing initiative defines a set of four properties at this level [Horn 2001]. This classification serves as the de facto standard in this domain. These properties have been defined in accordance to biological self-adaptation mechanisms [Kephart and Chess 2003]. For instance, the human body has similar properties in order to adapt itself to changes in its context (e.g., changing temperature in the environment) or self (an injury or failure in one of the internal organs). The following list further elaborates on the details.

- *Self-configuring* is the capability of reconfiguring automatically and dynamically in response to changes by installing, updating, integrating, and composing/decomposing software entities.
- *Self-healing*, which is linked to *self-diagnosing* [Robertson and Laddaga 2005] or *self-repairing* [de Lemos and Fiadeiro 2002], is the capability of discovering, diagnosing, and reacting to disruptions. It can also anticipate potential problems, and accordingly take proper actions to prevent a failure. Self-diagnosing refers to diagnosing errors, faults and failures, while self-repairing focuses on recovery from them.
- *Self-optimizing*, which is also called *self-tuning* or *Self-adjusting* [Sterritt et al. 2005], is the capability of managing performance and resource allocation in order to satisfy the requirements of different users. End-to-end response time, throughput, utilization, and workload are examples of important concerns related to this property.
- *Self-protecting* is the capability of detecting security breaches and recovering from their effects. It has two aspects, namely defending the system against malicious attacks, and anticipating problems and taking actions to avoid them or to mitigate their effects.

—**Primitive Level**: Self-awareness, self-monitoring, self-situated, and context-awareness are the underlying primitive properties [Horn 2001; Salehie and Tahvildari 2005a]. Some other properties were also mentioned in this level, such as openness [IBM-AC 2001] and anticipatory [Parashar and Hariri 2005], which are optional. The following list further elaborates on the details.

- *Self-Awareness* [Hinchey and Sterritt 2006] means that the system is aware of its *self* states and behaviors. This property is based on self-monitoring which reflects what is monitored.
- *Context-Awareness* [Parashar and Hariri 2005] means that the system is aware of its *context*, which is its operational environment.

2.2.2 *Relationships with Quality Factors.*

There is a belief that self-* properties are related to software quality factors. Salehie and Tahvildari discuss the potential links between these properties and quality factors [Salehie and Tahvildari 2005a]. The links can help us define and understand self-* properties better, and to utilize the existing body of knowledge on quality factors, metrics, and requirements, in developing and operating self-adaptive software. To demonstrate such relationships, it is better to analyze how a well-known set of quality factors defined in the ISO 9126-1 quality model [ISO/IEC 9126-1 2001] are linked to major and primitive self-* properties.

Self-configuring potentially impacts several quality factors, such as *maintainability*, *functionality*, *portability*, and *usability*. One may argue that self-configuring may be linked to reliability as well. This depends on the definition of reconfiguring. Assuming the definition given in the previous section (which is adopted by many researchers), we cannot associate all of the changes made to the system to for keep it reliable (e.g., fault recovery) with self-configuring. For self-healing, the main objective is to maximize the *availability*, *survivability*, *maintainability*, and *reliability* of the system [Ganek and Corbi 2003].

Self-optimizing has a strong relationship with *efficiency*. Since minimizing response time is often one of the primary system requirements, it also impacts *functionality*. On the other hand, self-protecting has a strong relationship with *reliability*, and it can also be linked to *functionality*. Primitive properties may also impact quality factors, such as *maintainability*, *functionality*, and *portability*. Sterritt *et al.* also emphasize this view by discussing the relationship between dependability aspects (e.g., availability and reliability) and the self-* properties [Sterritt and Bustard 2003].

2.3 Adaptation Requirements Elicitation

One plausible way to capture the requirements of self-adaptive software is getting help from the *six honest serving men*[1]. These six questions are very important in eliciting adaptation requirements. Laddaga uses a similar idea to partially address these requirements [Laddaga 2000]. The following set is a modified and completed version of Laddaga's questions to elicit the essential requirements of self-adaptive software.

[1]Six questions *What*, *Where*, *Who*, *When*, *Why* and *How*, called 5W1H, from "Six Honest Men" poem of R. Kipling, Just so stories. Penguin Books, London, 1902.

- **Where**: This set of questions are concerned with where the need for change is. Which artifacts at which layer (e.g., middleware) and level of granularity need to be changed? For this purpose, it is required to collect information about attributes of adaptable software, dependency between its components and layers, and probably information about its operational profile. Therefore, "where" questions set out to locate the problem that needs to be resolved by adaptation.
- **When**: Temporal aspects of change are addressed by this set of questions. When does a change need to be applied, and when it is feasible to do so? Can it be applied at anytime the system requires, or are there constraints that limit some types of change? How often does the system need to be changed? Are the changes happening continuously, or do they occur only as needed? Is it enough to perform adaptation actions reactively, or do we need to predict some changes and act proactively?
- **What**: This set of questions identifies what attributes or artifacts of the system can be changed through adaptation actions, and what needs to be changed in each situation. These can vary from parameters and methods to components, architecture style, and system resources. It is also important to identify the alternatives available for the actions and the range of change for attributes (e.g., parameters). Moreover, it is essential to determine what events and attributes have to be monitored to follow-up on the changes, and what resources are essential for adaptation actions?
- **Why**: This set of questions deals with the motivations of building a self-adaptive software application. As we discussed before, these questions are concerned with the objectives addressed by the system (e.g., robustness). If a goal-oriented requirements engineering approach is adopted to elicit the requirements, this set of questions identifies the goals of the self-adaptive software system.
- **Who**: This set of questions addresses the level of automation and human involvement in self-adaptive software. With respect to automation, it is expected that there will be minimum human intervention, whereas an effective interaction with system owners and managers is required to build trust and transfer policies (e.g. business policies). This issue will be discussed further in the taxonomy of adaptation (See Section 3).
- **How**: One of the important requirements for adaptation is to determine how the adaptable artifacts can be changed and which adaptation action(s) can be appropriate to be applied in a given condition? This includes how the order of changes, their costs and aftereffects are taken into account for deciding the next action/plan.

In order to realize self-adaptive software, the above questions need to be answered in two phases: a) the *developing phase*, which deals with developing and building self-adaptive software either from scratch or by re-engineering a legacy system, and b) the *operating phase*, which manages the operational concerns to properly respond to changes in the *self/context* of a software application. At the developing phase, designers elicit the requirements based on the above questions in order to build adaptable software as well as to set up mechanisms and alternatives to be used at the operating phase. At the operating phase, the system requires to adapt itself

based on the above questions. In fact, the questions in this phase are in general ask about "where" the source of need for a change is, what needs be changed, and when and how it is better to be changed. the answer to these questions at the operating phase depends on the approach and type of adaptation chosen at the developing phase. Some of these questions may be answered by administrators and managers through policies, and the rest should be determined by the system itself.

The distinction between the what and where questions is notable. "Where" addresses which part of the system caused the problem (e.g., deviation from quality goals), while "what" refers to the attributes and artifacts that need to be changed to resolve the problem. For example, in a multi-tier enterprise application, it is essential to know which part caused performance degradation (e.g., the database tier due to lack of resources) and after that, what needs to be changed (e.g., changing the service level at the web tier). Sometimes, the entity that is the source of change is also the entity that needs to be changed (e.g., component swapping). Therefore, although these questions are related, they address different aspects of adaptation.

2.4 Adaptation Loop

As explained earlier, self-adaptive software embodies a closed-loop mechanism. This loop, called the *adaptation loop*, consists of several processes, as well as sensors and effectors, as depicted in Fig. 2. This loop is called the MAPE-K loop in the context of autonomic computing, and includes the Monitoring, Analyzing, Planning and Executing functions, with the addition of a shared Knowledge-base[Kephart and Chess 2003]. Dobson *et al.* also represent the similar loop as autonomic control loop in the context of autonomic communication, including collect, analyze, decide and act [Dobson et al. 2006]. Oreizy *et al.* refer to this loop as adaptation management, which is composed of several processes for enacting changes and collecting observations, evaluating and monitoring observations, planning changes, and deploying change descriptions[Oreizy et al. 1999]. More details on the processes, sensors, and effectors of Fig. 2 are provided in the following sections.

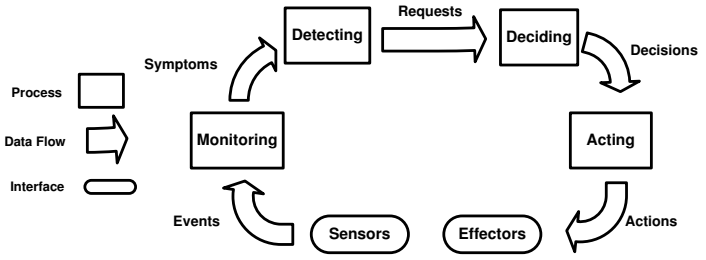

Fig. 2: Four Adaptation Processes in Self-Adaptive Software

2.4.1 *Adaptation Processes.*

The adaptation processes, which exist at the operating phase can be summarized as follows:

—The **monitoring process** is responsible for collecting and correlating data from sensors and converting them to behavioral patterns and symptoms. This process partly addresses the *where*, *when*, and *what* questions in the operating phase. The process can be realized through event correlation, or simply threshold checking, as well as other methods.

—The **detecting process** is responsible for analyzing the symptoms provided by the monitoring process and the history of the system, in order to detect *when* a change (response) is required. It also helps to identify *where* the source of a transition to a new state (deviation from desired states or goals) is.

—The **deciding process** determines *what* needs to be changed, and *how* to change it to achieve the best outcome. This relies on certain criteria to compare different ways of applying the change, for instance by different courses of action.

—The **acting process** is responsible for applying the actions determined by the deciding process. This includes managing non-primitive actions through predefined workflows, or mapping actions to what is provided by effectors and their underlying dynamic adaptation techniques. This process relates to the questions of *how*, *what*, and *when* to change.

2.4.2 *Sensors and Effectors.*

Sensors monitor software entities to generate a collection of data reflecting the state of the system, while effectors rely on in vivo mechanisms to apply changes. In fact, effectors realize adaptation actions. Sensors and effectors are essential parts of a self-adaptive software system. Indeed, the first step in realizing self-adaptive software is instrumenting sensors and effectors to build the *adaptable software*. Building adaptable software can be accomplished in an engineering or re-engineering manner. For example, Parekh *et al.* discuss adding sensors (probes) into legacy systems in order to retrofit the self-* properties [Parekh et al. 2006].

Table I lists the most common set of sensors and effectors in self-adaptive software. Logging is likely to be the simplest technique for capturing information from software. The logs need to be filtered, processed, and analyzed to mine significant information. The IBM Generic Log Adapter (GLA) and the Log Trace Analyzer (LTA) [IBM 2005] are examples of tools for this purpose.

Sensing and monitoring techniques from other areas can also be used. For instance, some of the protocols, standards, and formats that have been utilized are: CBE (Common Base Events) [IBM 2005], WBEM (Web-Based Enterprise Management) [WBEM] (containing CIM - Common Information Model [CIM]), and SIENA (Scalable Internet Event Notification Architectures) [Carzaniga et al. 2001]. Another noteworthy standard for sensing is ARM (Application Response Measurement) [ARM], which enables developers to create a comprehensive end-to-end management system with the capability of measuring the application's availability, performance, usage, and end-to-end response time. The ideas behind SNMP (Simple Network Management Protocol) [SNMP] for network and distributed systems are also applicable to self-adaptive software.

Profiling tools and techniques can also help in defining desirable sensors. The Java environment provides JVMTI (Java Virtual Machine Tool Interface) for this purpose [JVMTI]. Software management frameworks, such as JMX (Java Management eXtensions) [JMX] provide powerful facilities for both sensing and effecting.

Table I: Different Techniques for Realizing Sensors and Effectors

Entity	Technique	Example
Sensors	Logging	GLA (Generic Log Adapter), LTA (Log Trace Analyzer) [IBM 2005]
	Monitoring & events information models	CIM (Common Information Model) [CIM], CBE (Common Base Events) [IBM 2005]
	Management protocols and standards	Simple Network Management Protocol[SNMP] , Web-Based Enterprise Management[WBEM] , Application Response Measurement[ARM], Siena [Carzaniga et al. 2001]
	Profiling	JVMTI (JVM Tool Interface) [JVMTI]
	Management frameworks	JMX (Java Management eXtension) [JMX]
	Aspect-oriented programming	BtM (Build to Manage) [IBM BtM], JRat (Java Run-time Analysis Toolkit) [ShiftOne JRat]
	Signal monitoring	Heartbeat and pulse monitoring [Hinchey and Sterritt 2006]
Effectors	Design patterns	Wrapper (Adapter), Proxy, Strategy Pattern [Gamma et al. 1995]
	Architectural patterns	Microkernel pattern, Reflection pattern, Interception pattern [Buschmann et al. 1996; Alur et al. 2001]
	Autonomic patterns	Goal-driven self-assembly, self-healing clusters and utility-function driven resource allocation [Kephart 2005]
	Middleware-based effectors	Integrated middleware, Middleware interception [Popovici et al. 2002], Virtual component pattern [Schmidt and Cleeland 1999]
	Metaobject protocol	TRAP/J [Sadjadi et al. 2004]
	Dynamic aspect weaving	JAC [Pawlak et al. 2001], TRAP/J [Sadjadi et al. 2004]
	Function pointers	Callback in CASA [Mukhija and Glinz 2005]

Another notable idea along this line is pulse monitoring (reflex signal) [Hinchey and Sterritt 2006] adopted from Grid Computing, which is an extension of the heartbeat monitoring process. This technique encodes the status of the monitored entity.

Some of the effectors are based on a set of design patterns that allow the software system to change some artifacts during run-time. For instance, wrapper (adapter), proxy, and strategy are well-known design patterns [Gamma et al. 1995] for this purpose. Landauer *et al.* utilize the wrapping idea at the architecture level of adaptive systems [Landauer and Bellman 2001]. Moreover, microkernel, reflection, and interception are architectural patterns suitable for enabling adaptability in a software system [Buschmann et al. 1996; Alur et al. 2001]. Furthermore, Kephart mentions several design patterns, namely goal-driven self-assembly, self-healing clusters, and utility-function-driven resource allocation for self-configuring, self-healing, and self-optimizing [Kephart 2005], respectively. Babaoglu *et al.* also discuss design patterns from biology, such as plain diffusion and replication, which are applicable to distributed adaptable systems [Babaoglu et al. 2006].

An important class of techniques for effectors is based on middleware. In these solutions, system developers realize effectors at the middleware layer by intercepting the software flow [Popovici et al. 2002], or by using design patterns [Schmidt and Cleeland 1999]. Other solutions have been proposed for implementing effectors using dynamic aspect weaving (e.g., in JAC [Pawlak et al. 2001]), metaobject pro-

tocol (e.g., in TRAP/J [Sadjadi et al. 2004]), and the function pointers technique (e.g., in CASA [Mukhija and Glinz 2005] for realizing callback). Some middlewares provide support for dynamic aspect weaving. For example, JBoss has an AOP module, with the capability of dynamic weaving (e.g., used for effectors in [Salehie et al. 2009]).

3. A TAXONOMY OF SELF-ADAPTATION

Several works have already discussed different aspects of self-adaptation. Oreizy *et al.* discuss the spectrum of self-adaptivity, which generally starts from static approaches and then moves on to dynamic ones [Oreizy et al. 1999]. On the other hand, McKinley *et al.* focus more on the techniques and technologies in this domain [McKinley et al. 2004]. This article unifies these classifications into a taxonomy, and also introduces new facets to fill in the gaps.

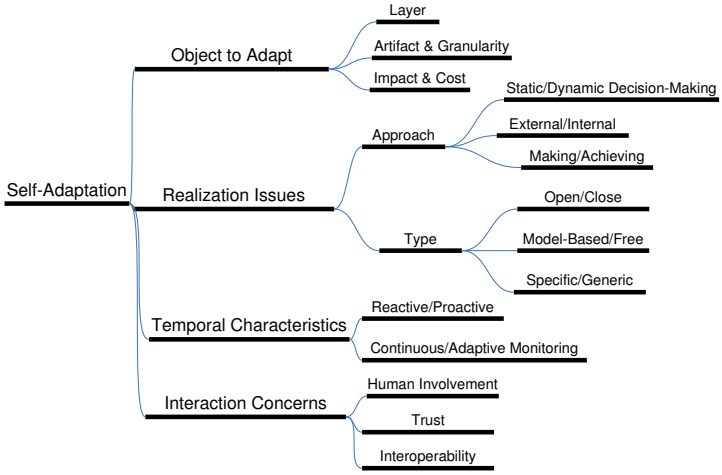

Fig. 3: Taxonomy of Self-Adaptation

Figure 3 illustrates the hierarchy of the introduced taxonomy. The first level includes: "object to adapt", "realization issues", "temporal characteristics" and "interaction concerns". In organizing these facets, we kept the requirements questions, introduced in Section 2.3, in mind. Although these facets cannot be mapped to the questions on a one-to-one basis, one or two questions in each facet are emphasized: "object to adapt" mainly addresses the "what" and "where" questions, while "realization issues" deal more with the "how" concerns, and "temporal characteristics" deal with issues related to the "when" aspect. Interaction concerns are more or less related to all four of where-when-what-how questions as well as to the "who" question.

3.1 Object to Adapt

This facet of the proposed taxonomy deals with *where* and *what* aspects of the change. In fact, both sets of what and where questions are covered in the developing

and operating phases.

—*Layer*: Which layer of the software system (i.e., *where*) is not as expected, based on the requirements? Which layer of the system can be changed and needs to be changed? Adaptation actions can be applied to different layers. McKinley *et al.* [McKinley et al. 2004] define two levels of application and middleware for this purpose, in which middleware is decomposed into four sub-layers, as described in [Schmidt 2002]. Application-layer adaptation differs from middleware-layer adaptation in a number of ways. For example, in application-layer adaptation, changes usually have direct impact on the user, and consequently, they may require the user's explicit approval and trust [Lapouchnian et al. 2005].

—*Artifact and Granularity*: Which artifact(s) and at which level of granularity can/needs to be changed? *What* artifact, attribute, or resource can/needs to be changed for this purpose? Adaptation can change the modules or the architecture and the way they are composed. An application can be decomposed into services, methods, objects, components, aspects, and subsystems depending on the architecture and technologies used in its implementation. Each of these entities, as well as their attributes and compositions, can be subject to change, and therefore, adaptation can be applied in fine or coarse levels of granularity.

—*Impact & Cost*: The impact describes the scope of aftereffects, while cost refers to the execution time, required resources, and complexity of adaptation actions. This facet is related to *what* the adaptation action will be applied to, and partly to *how* it will be applied.

Based on both impact and cost factors, adaptation actions can be categorized into the *weak* and *strong* classes. Weak adaptation involves modifying parameters (parameter adaptation) or performing low-cost/limited-impact actions, whereas strong adaptation deals with high-cost/extensive-impact actions, such as replacing components with those that improve system quality [McKinley et al. 2004]. Generally, weak adaptation includes changing parameters (e.g., bandwidth limit), using pre-defined static mechanisms (e.g., load-balancing), or other actions with local impact and low cost (e.g., compressing data). Strong adaptation may change, add, remove, or substitute system artifacts. Cost in this classification refers to how much time and resources an action would need. It also highly depends on whether the action's requirements (e.g., alternatives for switching) are ready, or will become ready at run time. Table II lists a typical set of adaptation actions in both of the weak and strong categories.

It is notable that this classification is not the same as "artifact & granularity", even though one may argue that in general, higher levels (e.g., architecture) have higher cost and impact. Although in some cases there may be some correlation between granularity and cost/impact, this is not always the case. An example is the case of having a load-balancing action that routes requests through duplicate components or servers. Another noteworthy point is that strong actions are mostly composite, and may contain several weak/strong actions. For example, changing the architecture may require redeployment of some components and changing a few parameters.

Table II: Typical Weak/Stong Adaptation Actions in Self-Adaptive Software

Type	Action	Description
Weak	Caching [Oreizy et al. 1999; Dowling and Cahill 2004]	Caching data, states, connections, objects or components in order to lower the response time, load of servers, or help decentralized management
	Changing data quality [Mukhija and Glinz 2005]	Changing data quality (i.e., lower resolution) to save bandwidth and increase speed
	Changing type of data [Cheng et al. 2006; Mukhija and Glinz 2005]	For instance, switching from video to image and even to text to save bandwidth and increase speed
	Compressing data [Laddaga et al. 2001]	Saving bandwidth by transceiving compressed data
	Tuning (parameter adjusting) [Karsai et al. 2001]	Adjusting parameters to meet some adaptation goals (i.e., buffer size and delay time)
	Load balancing [Willebeek-LeMair et al. 1993; Cardellini et al. 1999]	Fair division of load between system elements to achieve maximum utilization, throughput or minimum response time
	Changing aspects [Pinto et al. 2002; Suvée et al. 2003]	Changing aspect of a component or object with another one with different quality
	Changing algorithm/ method [Oreizy et al. 1999; Robertson and Williams 2006]	Changing the algorithm/ method to meet self-* properties and run-time constraints
Strong	Replacement, addition & removal [McKinley et al. 2004]	Replacing an entity (e.g., a component) by another one with the same interface but different quality (non-functional)
	Restructuring /changing architecture [Kramer and Magee 1990; Magee and Kramer 1996; Oreizy et al. 1998]	Changing organization/ architecture of the system (it may change the architectural style or design patterns of the system)
	Resource provisioning [Appleby et al. 2001]	Provisioning additional resources at different levels (this action can be extended to adding/removing any resources, such as servers)
	Restarting / redeployment [Candea et al. 2006]	Restarting/ rebooting (macro- or micro-) or redeployment of system entities at different levels mainly due to faults/failures

3.2 Realization Issues

This facet of the proposed taxonomy deals with *how* the adaptation can/needs to be applied. These issue are categorized into *approach* and *type* classes, and are discussed further in the following.

3.2.1 *Adaptation Approach.*

One significant facet of the taxonomy is the approach of incorporating adaptivity into the system. The following sub-facets can be identified:

—*Static/Dynamic Decision-Making*: This sub-facet specifically deals with how the deciding process can be constructed and modified. In the static option, the deciding process is hard-coded (e.g., as a decision tree) and its modification requires recompiling and redeploying the system or some of its components. In dynamic decision-making, policies [Kephart and Walsh 2004], rules [Liu et al. 2004] or QoS definitions [Loyall et al. 1998] are externally defined and managed,

so that they can be changed during run-time to create a new behavior for both functional and non-functional software requirements.

—*External/Internal Adaptation:* From a different perspective, the adaptation can be divided into two categories with respect to the separation of the adaptation mechanism and application logic. These two categories, as depicted in Fig. 4, are as follows:

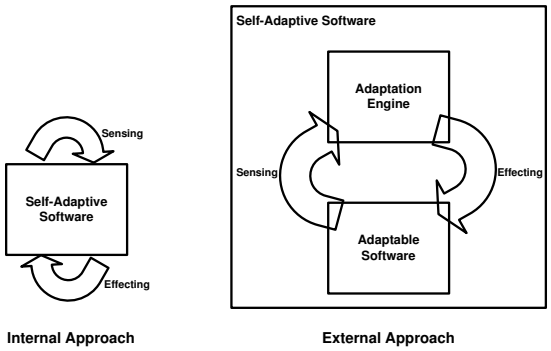

Fig. 4: Internal and External Approaches for Building Self-Adaptive Software

- *Internal* approaches intertwine application and the adaptation logic. This approach is based on programming language features, such as conditional expressions, parametrization, and exceptions [Oreizy et al. 1999; Floch et al. 2006]. In this approach the whole set of sensors, effectors, and adaptation processes are mixed with the application code, which often leads to poor scalability and maintainability. This approach can be useful for handling local adaptations (e.g., for exception handling). However, adaptation often needs global information about the system and correlating events happening in its self/context. Generally, this approach may be realized by extending existing programming languages or defining new adaptation languages.
- *External* approaches use an external *adaptation engine* (or manager) containing adaptation processes. As depicted in Fig. 4, using this approach, the self-adaptive software system consists of an adaptation engine and an *adaptable software*. The external engine implements the adaptation logic, mostly with the aid of middleware [Kon et al. 2002; Floch et al. 2006], a policy engine [Bigus et al. 2002], or other application-independent mechanisms. In a complex and distributed system, it is quite common to have multiple self-adaptive *element*s, each containing these two parts. In this case, the composition of elements in an appropriate architecture and an infrastructure for interoperability are essential.

The internal approach has some notable drawbacks. For instance, in this case the system will be costly to test and maintain/evolve, and it is often not scalable. On the other hand, a significant advantage of the external approach is the reusability of the adaptation engine, or some realized processes for various applications. This means that an adaptation engine can be customized and configured for different systems.

—*Making/Achieving Adaptation*: Generally speaking, self-adaptivity can be introduced into software systems using two strategies [Horn 2001]. The first strategy is to engineer self-adaptivity into the system at the developing phase. The second strategy is to achieve self-adaptivity through adaptive learning. Sterritt [Sterritt 2003] calls these two approaches *making* and *achieving*. *Making* has an implied system and/or software engineering view to engineer adaptivity into the individual systems. *Achieving* has an implied artificial intelligence and adaptive learning view to achieve adaptive behavior. These two approaches do not necessarily contradict each other in the sense that their combination can be utilized as well.

3.2.2 *Adaptation Type.*

Another important facet is the type of adaptation. It specifies whether the adaptation is open or closed to new alternatives, whether it is domain specific or generic, and whether it is model-based or model-free.

—*Close/Open Adaptation*: A close-adaptive system has only a fixed number of adaptive actions, and no new behaviors and alternatives can be introduced during run-time. On the other hand, in open adaptation, self-adaptive software can be extended, and consequently, new alternatives can be added, and even new adaptable entities can be introduced to the adaptation mechanism (e.g., through new joint-points for weaving aspects [Pawlak et al. 2001]).

—*Model-Based/Free Adaptation*: In model-free adaptation, the mechanism does not have a predefined model for the environment and the system itself. In fact, by knowing the requirements, goals, and alternatives, the adaptation mechanism adjusts the system. For example, Dowling uses model-free Reinforcement Learning (RL) in adaptation [Dowling 2004]. On the other hand, in model-based adaptation, the mechanism utilizes a model of the system and its context. This can be realized using different modeling approaches, such as a queueing model for self-optimizing [Litoiu et al. 2005], architectural models for self-healing [Garlan and Schmerl 2002], or domain-specific models in [Karsai and Sztipanovits 1999].

—*Specific/Generic Adaptation*: Some of the existing solutions address only specific domains/applications, such as a database (e.g., IBM SMART project [IBM SMART]). However, generic solutions are also available, which can be configured by setting policies, alternatives, and adaptation processes for different domains (e.g., Accord [Liu et al. 2004]). This type addresses *where* and *what* concerns in addition to *how*, because the specific type only focuses on an adaptation of artifacts or attributes of a particular part of the software system.

3.3 Temporal Characteristics

This facet deals with issues regarding *when* artifacts can/need to be changed. The following sub-facets can be identified:

—*Reactive/Proactive Adaptation:* This sub-facet captures the self-adaptive software anticipatory property [Parashar and Hariri 2005]. In the reactive mode, the system responds when a change has already happened, while in the proactive mode, the system predicts when the change is going to occur. This issue impacts the detecting and the deciding processes.

—*Continuous/Adaptive Monitoring:* This sub-facet captures whether the monitoring process (and consequently sensing) is continually collecting and processing data vs. being adaptive in the sense that it monitors a few selected features, and in the case of finding an anomaly, aims at collecting more data. This decision affects the cost of the monitoring and detection time.

3.4 Interaction Concerns

One cannot discuss the taxonomy without addressing the issue of interaction with other self-adaptive software systems/elements through interfaces. This facet consists of interacting with humans and/or other elements/systems. The facet is related to all four of the where-when-what-how questions as well as the "who" question. The following sub-facets can be identified:

—*Human Involvement*: As noted before, this facet is related to the question of "who" the agent of change is. In self-adaptive software, human involvement can be discussed from two perspectives. First, the extent to which the mechanism is automated, and second, how well it interacts with its users and administrators. For the former, we can use the maturity model proposed in autonomic computing [Murch 2004]. The levels in this model include basic, managed, predictive, adaptive and autonomic. According to this view, human involvement is not desirable, therefore more automation is demanding. However, the second view addresses the quality of human interaction to either express their expectations and policies, or to observe what is happening in the system. According to this view, human involvement is essential and quite valuable for improving the manageability and trustworthiness of self-adaptive software. These issues have been addressed in some earlier research, such as [Oreizy et al. 1999] and [Kephart 2005]. For the rest of this article, the second meaning will be used.

—*Trust*: Trust is a relationship of reliance, based on past experience or transparency of behavior. One view of trust is security, as highlighted by Dobson *et al.* in autonomic systems [Dobson et al. 2006]. Another view, not orthogonal to the first one, is related to how much human or other systems can rely on self-adaptive software systems to accomplish their tasks. This view is linked first to assurance and dependability. Georgas *et al.* relate this issue to dependability as the "extent to which a system can be trusted to appropriately and correctly adapt" [Georgas et al. 2005]. However, as McCann *et al.* point out, trust is not necessarily based on self-adaptive services or its quality [McCann et al. 2006; Huebscher and McCann 2008]. They discuss how trust can be built via revealing significant information about the system status and the visibility of adaptation processes. This explains why predictability can be considered as a major factor for placing trust upon self-adaptive software.

—*Interoperability Support*: Self-adaptive software often consists of elements, modules, and subsystems. Interoperability is always a concern in distributed complex systems for maintaining data and behavior integrity across all constituent elements and subsystems. In self-adaptive software, the elements need to be coordinated with each other to have the desired self-* properties and to fulfill the expected requirements. Global adaptation requirements will be met if elements and designated mechanisms in different layers and platforms of a system

are interoperable (e.g., middleware and application).

4. A LANDSCAPE OF THE RESEARCH AREA

This section presents a landscape of current research in self-adaptive software systems. Two broad aspects of such a landscape are discussed: i) supporting disciplines, and ii) a selected number of projects. The former points out how various disciplines dealing with adaptive systems can support building, evaluating, and utilizing self-adaptive software systems, while the latter gives a picture of the current status in this area.

4.1 Supporting Disciplines

The first viewpoint of the landscape shows how different disciplines are able to support and contribute to developing and operating self-adaptive software systems. The noteworthy point is that self-adaptive software is inherently interdisciplinary, and the combination of disciplines highly depends on the design metaphors adopted for building a specific self-adaptive software system. Laddaga enumerates three such design metaphors used by early researchers: coding an application as a dynamic planning system, coding an application as a control system [Laddaga 1999], and coding a self-aware system [Laddaga 2000]. These metaphors utilize ideas from artificial intelligence, control theory and decision theory. The following sections discuss several such disciplines - namely *software engineering*, *artificial intelligence*, *decision theory*, *control theory*, and *network and distributed computing*. Although several other disciplines, such as *optimization theory*, can also be added to this list, due to space limitations, they are only partially discussed in connection with the other disciplines.

4.1.1 *Software Engineering.*

Numerous research areas in software engineering are related to self-adaptive software. As discussed in Section 2.2.2, self-* properties can be related to quality factors. Consequently, the ideas developed in the context of *software quality* for realizing and measuring quality (including all -ilities) are potentially applicable to self-adaptive software. A few research efforts, see for example [Salehie and Tahvildari 2007], have aimed to establish this link. The important point is that self-* properties are mostly related to Non-Functional Requirements (NFR), such as security and performance. In fact, fulfilling these requirements is the major trigger for change. However, functional requirements are also related to self-adaptation. An example is changing a component's behavior to an acceptable lower level of functionality, in order to improve its performance. These issues also bring *requirements engineering* into the picture. Several researchers have used NFR models, particularly goal models, in self-adaptive software; see for example [Lapouchnian et al. 2005] and [Subramanian and Chung 2001].

Coupling software with its specification and formal model can allow monitoring correctness and many other metrics with respect to the specified requirements and self-* properties [Pavlovic 2000]. *Formal methods* provide various ways for modeling software systems as well as utilizing such models. Accordingly, it is possible to rely on formal methods to model adaptable software in adaptation processes. Moreover, formal methods can be used for validation and verification of self-adaptive software

to ensure its correct functionality, and to understand its behavior [Laddaga et al. 2000]. Due to various differences between traditional and self-adaptive software, the existing models and methods developed for non-adaptive software systems are not directly applicable. This means that new approaches based on formal models, such as Model-Integrated Computing (MIC) [Karsai and Sztipanovits 1999], are required for this purpose. MIC has been applied successfully to some domain-specific embedded systems for managing run-time re-configuration [Scott et al. 2000].

Software Architecture models and languages, such as Architectural Description Languages (ADL), can certainly be helpful in software modeling and management, particularly at run-time. Bradbury *et al.* survey several ADLs based on graphs, process algebras, and other formalisms for dynamic software architectures [Bradbury et al. 2004]. Garlan *et al.* use Acme ADL to describe the architecture of adaptable software and to detect violations from defined constraints [Garlan et al. 2004]. Oreizy *et al.* point out that software architecture can also help in change management [Oreizy et al. 1999]. Another idea that can be useful is Attribute-Based Architecture Styles (ABAS) [M.H. Klein and Lipson 1999] as an extension of architectural styles. In fact, ABAS includes a quality-attribute specific model (e.g., performance), in order to provide a method of reasoning about an architecture design and the behavior of its interacting component types.

Component-Based Software Engineering (CBSE) can help the development of self-adaptive software in two ways. First, it is easier to design and implement an adaptable software relying on component models. Second, an adaptation engine needs to be modular and reusable, and CBSE can also be used in its development. Moreover, as pointed out in ACT [Sadjadi and McKinley 2004], component models can be used in adaptive systems as a means of incorporating the underlying services for dynamic adaptation and adaptation management. Another related area, *Aspect-Oriented Programming (AOP)* and more specifically dynamic AOP, can also be used in realizing self-adaptive software. This facilitates encapsulating adaptation concerns in the form of aspects through dynamic run-time adaptation. It also helps in implementing fine-grained adaptation actions at a level lower than components [Greenwood and Blair 2004; Salehie et al. 2009]. For example, JAC (which is a dynamic AOP framework [Pawlak et al. 2001]) uses a wrapping chain that can dynamically change existing or new joint points. AOP can also be used for instrumenting sensors as in the IBM BtM (Build to Manage) tool [IBM BtM].

Service Computing and *Service-Oriented Architecture* (SOA) can also support realizing self-adaptive software by facilitating the composition of loosely coupled services. Web service technology is often an appropriate option for implementing dynamic adaptable business processes and service-oriented software systems, due to their flexibility for composition, orchestration, and choreography [Peltz 2003]. Birman *et al.* propose extensions to the web services architecture to support mission-critical applications [Birman et al. 2004]. Examples include standard services that track the health of components, mechanisms for integrating self-diagnosis into applications, and automated configuration tools. Another notable work is Autonomic Web Processes (AWP) [Verma and Sheth 2005], which are web service-based processes that support the self-* properties.

4.1.2 *Artificial Intelligence.*

As noted by Laddaga [Laddaga 1999], in general, it is surprising that not much has been done to apply Artificial Intelligence (AI) techniques, such as planning and probabilistic reasoning, to develop and manage software systems. In particular, self-adaptive software has a remarkable common ground with AI and adaptive systems (see for example [Sterritt et al. 2005]). For the detecting process, AI can assist in log/trace analysis and pattern/symptom matching to identify abnormal conditions or the violation of constraints. AI is also rich in planning, reasoning, and learning, which could be useful in the deciding process. One obstacle is quality assurance for AI-based systems, which becomes necessary because of the utilized intelligent, heuristic, and search-based techniques. This issue was pointed out by Parnas about two decades ago a specific class of software systems, but it is generalizable to the other systems as well [Parnas 1985].

One interesting approach to realize self-adaptive software is based on *AI planning*. In this approach, a software system plans and may replan its actions instead of simply executing specific algorithms. This is in particular related to the deciding process for selecting the appropriate course of action. The *planning-based adaptation* should be active all the times, through searching among existing plans or by composing adaptation actions. In fact, the adaptation engine needs *continuous planning* via *contingency planning* or *replanning* [Russell and Norvig 1995]. The former provides a conditional plan with alternative paths based on the sensed information, while the latter generates an alternative plan in the case that the original plan fails. A notable point is that planning, at least in its basic form, cannot be used for all of the self-* properties. According to [Srivastava and Kambhampati 2005], among all of the self-* properties, planning has the highest potential for being used in conjunction with self-healing. One example of using AI-planning for self-healing is the work by Arshad *et al.* [Arshad et al. 2003].

An important concept that can be used in self-adaptive software is the way *software agents* model their domains, goals, and decision-making attributes. For example, an interesting goal-based model for action selection (composed of actions and goals with activation levels) has been proposed by Maes [Maes 1990]. Goal-oriented requirements modeling is a well established area of research in agent-based systems and there are many research efforts that involve these models and methods in self-adaptive software, including [Lapouchnian et al. 2005] and [Morandini et al. 2008].

The other important issues, especially in *Multi-Agent Systems* (MAS), are coordination models and distributed optimization techniques, which can be useful in multi-element self-adaptive software; for example, see [Bernon et al. 2003]. In such systems, local and global goals, which are mostly derived from self-* properties, need to be coordinated. Tesauro *et al.* realize a sample MAS, called Unity, as a decentralized autonomic architecture based on multiple interacting agents [Tesauro et al. 2004]. Weyns *et al.* have also investigated employing multi-agent architectures for self-adaptive software. For example, they utilized a situated multi-agent architecture for a self-adaptive automated guided vehicle transportation system [Weyns et al. 2005].

Machine Learning and *Soft Computing* are other areas with the potential to play important roles in self-adaptive software, especially through the "achieving" ap-

proach. Achieving needs analyzing the stream of sensed data and learning the best way to act. Oreizy *et al.* name evolutionary programming and AI-based learning in the category of approaches dealing with unprecedented changes with a clearer separation of concerns [Oreizy et al. 1999]. These algorithms generally use properties of the environment and knowledge gained from previous attempts to generate new algorithms. Genetic algorithms and different on-line learning algorithms, such as Reinforcement Learning (RL) can also be used for this purpose. RL is a promising option for dynamic action selection [Amoui et al. 2008] and decentralized collaborative self-adaptive software [Dowling 2004]. Tesauro discusses that RL has the potential to achieve better performance as compared to traditional methods, while requiring less domain knowledge [Tesauro 2007]. He also adds that as RL relies on exploration for training, it is not always feasible to learn policies in a live system. Instead, offline training or a hybrid method to use the existing policies should be utilized. Fuzzy logic is also applicable to address the fuzziness of quality goals and policies [Salehie and Tahvildari 2005b].

Another notable field, related to this context under the umbrella of artificial intelligence is *decision theory*. This theory, in both classical and qualitative forms, can contribute to realizing the deciding process. The classical form is suitable for cases in which decision-making relies on maximizing a certain utility function in a deterministic manner, while the qualitative form is appropriate for problems including uncertainty.

One of the areas applicable to self-adaptive software is *utility theory*. The term *utility* refers to "the quality of being useful" for an action, choice, or alternative [Russell and Norvig 1995], and can be identified either with certainty or with uncertainty (in classical or qualitative form). Therefore, utility theory deals with methods to assign an appropriate utility value to each possible outcome and to choose the best course of action based on maximizing the utility value [Keeney and Raiffa 1976]. Walsh *et al.* demonstrate how utility functions can enable autonomic elements to continually optimize the use of computational resources in a dynamic and heterogeneous environment (a data center prototype) [Walsh et al. 2004]. Poladian *et al.* employ a utility function for user needs and preferences in resource-aware services [Poladian et al. 2004]. They use this function to formulate an optimization problem for dynamic configuration of these services. Nowicki *et al.* deal with decentralized optimization using utility functions [Nowicki et al. 2005]. Boutilier *et al.* use utility functions in self-optimizing and rely on incremental utility elicitation to perform the necessary computations using only a small set of sampled points from the underlying function [Boutilier et al. 2003; Patrascu et al. 2005].

In practice, due to uncertainty, probabilistic reasoning and *decision-theoretic planning* are required in decision making. Markov Decision Process (MDP) and Bayesian network are two well-established techniques for this purpose. These techniques are also applicable to realizing self-* properties due to their uncertain attributes. For example, there are several research efforts utilizing these models for diagnosis and self-recovery, for example see [Howe 1995], [Robertson and Laddaga 2005] and [Robertson and Williams 2006]. Porcarelli *et al.* also use a stochastic Petri net for decision-making in fault-tolerance [Porcarelli et al. 2003].

4.1.3 *Control Theory/Engineering.*

Control theory/engineering, similar to self-adaptive software, is concerned with systems that repeatedly interact with their environment through a sense-plan-act loop. The notions of adaptation and feedback have been discussed for decades in control theory and engineering, and have been utilized in designing and developing systems in various domains. The control-based paradigm considers the software system (adaptable software) as a *controllable plant* with two types of inputs: control inputs, which control the plant's behavior, and disturbances, which change the plant's behavior in an unpredictable manner [Kokar et al. 1999]. A *controller* (adaptation engine) changes the values of the plant's control inputs.

The control-based paradigm is often based on a model of the software plant's behavior. For instance, Litoiu *et al.* use a hierarchical Layered Queue Model (LQM) of a software system for tuning parameters (weak adaptation) [Litoiu et al. 2005]. Abdelwahed *et al.* [Abdelwahed et al. 2004] also show that a model-based control architecture can realize the self-optimizing property, by tuning the plant parameters. Moreover, Bhat *et al.* discuss applying online control models to achieve self-managing goals by extending the ACCORD component framework [Bhat et al. 2006]. Although closed-loop is the most widely used model for control-based self-adaptive software, adaptive and reconfigurable models are also recommended for several reasons, including large-range dynamic disturbances [Kokar et al. 1999]. On the other hand, considering the discrete nature of software systems, one of the appropriate control-based approaches for self-adaptive software is *supervisory control* of *discrete event system* (DES) [Ramadge and Wonham 1987]; see for example [Tziallas and Theodoulidis 2004] and [Karsai et al. 2001].

Traditionally, control theory has been concerned with systems that are governed by the laws of physics. This allows them to make assertions about the presence or absence of certain properties, which is not necessarily the case with software systems. In practice, checking software controllability or building a controllable software is a challenging task, often involving non-intuitive analysis and system modifications [Karamanolis et al. 2005]. Therefore, some researchers believe that applying this approach to software is often more complex than the case of traditional control systems [Robertson and Laddaga 2005].

4.1.4 *Network and Distributed Computing.*

Techniques used in network and distributed computing can be extensively applied to self-adaptive software. This is due to the fact that the bulk of the existing software systems are distributed and network-centric. Although it may be difficult to directly apply some of these techniques to all layers of self-adaptive software (i.e., policy management at the application layer), their usage in addressing adaptation requirements and the engineering of such systems is promising. Another line of research in this area concerns Peer-to-Peer (P2P) applications and ad hoc networks, which deal with the dynamic change of environment, architecture, and quality requirements. Research in this area often uses self-organizing elements in a bottom-up approach. However, as explained earlier, this article does not cover systems with the self-organizing property.

Policy-based management is one of the most successful approaches followed in network and distributed computing [Sloman 1994]. Policy-based management specifies how to deal with situations that are likely to occur (e.g., priorities and access con-

trol rules for system resources). Most of the definitions given for policy emphasize on providing guidance in determining decisions and actions. The policy management services normally consist of a policy repository, a set of Policy Decision Points (PDP) for interpreting the policies, and a set of Policy Enforcement Points (PEP) for applying the policies [Westerinen et al. 2000]. The most widely used policy type in networks is the *action policy* (in the form of event-condition-action rules) which is also applicable to self-adaptive software. In addition, other policy types like *goal policy* (specifying a desired state), and *utility policy* (expressing the value of each possible state) can also be exploited in self-adaptive software [Kephart and Walsh 2004]. The adaptation policies may need to be changed based on new requirements or conditions. Some research efforts have addressed this issue. For example, Lutfyyia *et al.*, among several other efforts on policy-based management, have proposed a control-theoretic technique for dynamic change of policies in a data center [Aly and Lutfiyya 2007]. Policy-based management has been adopted in a number of self-adaptive software research; see for example [Keeney and Cahill 2003; Badr et al. 2004; Salehie et al. 2009]. Some frameworks have also been introduced for developing policy-based self-adaptive software, e.g., StarMX framework in the domain of enterprise Java-based systems [Asadollahi et al. 2009].

Quality of Service (QoS) management, another successful area in networking and distributed systems [Hutchison et al. 1994], is closely related to policy management [Lutfiyya et al. 2001]. QoS requirements are related to non-functional requirements of a system, and consequently, they can be linked to self-* properties in distributed software systems. In this context, QoS management methods rely on either modeling the application, e.g., queuing models, or using well-understood components, e.g., Prediction-Enabled Component Technology (PECT) [Woodside and Menascé 2006]. Therefore, QoS management can assist in modeling the quality factors of a self-adaptive software system (and consequently self-* properties), and also in realizing adaptation processes.

Another powerful technology borrowed from distributed and network-based systems is *middleware*. Middleware-based adaptation (in all four sub-layers discussed by Schmidt [Schmidt 2002]) would also be applicable to adaptation processes. For instance, generic components of decision-making and change detection can be realized at the middleware level; for example, see [Floch et al. 2006] and [Kumar et al. 2007].

One of the well established areas in networks and distributed systems is *resource management*. In specific, *virtualization* techniques can have a significant impact on the quality of self-adaptive software. Virtualization reduces the domain of an adaptation engine to the contents of a virtual machine [Menascé and Bennani 2006]. Consequently, dynamic resource management and resource provisioning are easier to handle. Virtualization also provides an effective way for legacy software systems to coexist with current operational environments [Bantz et al. 2003]. This property can be utilized in building adaptable software from legacy systems.

Monitoring and sensing techniques have been widely used in networks and distributed systems. Basic techniques like heartbeat monitoring and more advanced techniques like pulse monitoring have been used in self-adaptive and self-managing software [Hinchey and Sterritt 2006]. One important issue, which is quite signif-

icant in self-adaptive software, is the cost of sensing and monitoring. This issue has been addressed extensively in networking (e.g., [Dilman and Raz 2002]) and distributed systems (e.g., [Saab et al. 2002]).

4.2 Research Projects

The projects in this section are selected from different academic and industrial sectors to capture main research trends in the broad area of self-adaptive software. We collected information from many academic and industrial research projects (from IBM, HP, Microsoft, Sun, etc). However, we selected a few of them to represent the major research ideas in this field. Space limitations, the diversity of ideas, and their impact on the field, are the concerns taken into account for the selection.

Among other goals, the discussions in this section aim to identify the existing research gaps in this area. For this purpose, the projects are analyzed from several points of view. Since some of these projects are not available for evaluation, the reported properties and features are based on the referenced material.

Table III lists the selected projects sorted based on the date of cited publication(s). In the case of several related publications from the same research group, the more important ones are cited. These projects are selected on the basis of their impact on the area and the novelty/significance of their approach. In the rest of this section, we compare the selected projects in relation to three different views, namely *self-* properties*, *adaptation processes*, and the proposed *taxonomy facets*.

The first view discusses the major self-* properties that are supported by each project, as shown in Table IV. We can see that the majority of these projects focus on one or two of the known self-* properties. This shows that the coordination and the orchestration among multiple self-* properties have not yet received the full attention they deserve. Another notable point is that a limited number of projects in the literature support self-protecting (only one project in the selected set). Generally speaking, this is due to constant changes in the network topology, the increasing variety of software components/services, and the increasing complexity, as well as variety of attacks and viruses [Qu and Hariri 2007]. Most of the research dealing with the self-protecting property focus on the network layer, and particularly, on detecting attacks. Such research efforts are outside of the main scope of this article, as we have already mentioned.

The second view is concerned with how the selected projects address adaptation processes. Table V compares and categorizes the selected projects according to four levels from "no support" to "high support". The level for each process is determined based on efficiency, coverage of different aspects, and support for available standards. Each of the processes also has its own specific aspects. For example, to evaluate the *deciding process*, we investigate whether a project takes into account dynamicity and uncertainty. To analyze the table column-wise, we have included a vector with four components reflecting the relative frequencies of the different levels in each column, ranging from "no support" (−) to "high support" (H). For example, the vector $(2, 5, 7, 2)/16$ under the *Monitoring* column shows that out of the 16 selected projects, there are 2 with "no support", 5 with "low support", 7 with "medium support", and 2 with "high support". The column-wise assessment shows that monitoring, detecting, deciding, and acting each have only 2 or 3 projects with "high support" out of the selected 16 . These observations indicate that one

Table III: Selected Projects in the Area of Self-adaptive Software

Projects	Summary
Quo [Loyall et al. 1998]	Quality Objects (QuO) provides Quality Description Languages (QDL) for specifying possible QoS states, the system resources and mechanisms for measuring and controlling QoS, and behavior for adapting to changing levels of available QoS at run-time.
IBM Oceano [Appleby et al. 2001]	Developing a pilot prototype of a manageable infrastructure for a computing utility powerplant
Rainbow [Garlan et al. 2004; Garlan and Schmerl 2002]	Proposing an architecture-based adaptation framework consisting of an adaptation infrastructure and a system-specific adaptation knowledge
Tivoli Risk Manager [Tuttle et al. 2003]	Providing an integrated security management structure by filtering and correlating the data from different sources and applying dynamic policies
KX [Kaiser et al. 2003; Valetto and Kaiser 2003]	A generic framework for collecting and interpreting application-specific behavioral data at run-time through sensors (probes) and gauges
Accord [Liu et al. 2004]	Providing a programming framework for defining application context, autonomic elements, rules for the dynamic composition of elements, and an agent infrastructure to support rule enforcement
ROC [Candea et al. 2004; Candea et al. 2006]	Building Recursively Recoverable (RR) systems, based on micro-reboot, online verification of recovery mechanisms, isolation and redundancy, and system-wide support for undo
TRAP [Sadjadi et al. 2004; Sadjadi and McKinley 2004]	A tool for using aspects and reflective technique for dynamic adaptation in Java, TRAP/J, and .Net framework, TRAP/.Net
K-Component [Dowling and Cahill 2004; Dowling 2004]	A meta-model for realizing a dynamic software architecture based on *Adaptation Contract Description Language* (ACDL) for specifying reflective programs. ACDL separates the specification of a system's self-adaptive behavior from the system components' behavior
Self-Adaptive [Robertson and Laddaga 2005]	Establishing a model-based diagnosis and automatic recovery approach using of method deprecation and regeneration with the aid of a decision-theoretic method dispatch
CASA [Mukhija and Glinz 2005]	Contract-based Adaptive Software Architecture (CASA) supports both application-level and low-level (e.g., middleware) adaption actions through an external adaptation engine
J3 [White et al. 2005]	Providing a model-driven framework for application-level adaptation based on three modules J2EEML, JAdapt, and JFense respectively for modeling, interpreting and run-time management of self-adaptive J2EE applications
DEAS [Lapouchnian et al. 2005]	Proposing a framework for identifying the objectives, analyzing alternative ways of how these objectives can be met, and designing a system that supports all or some of these alternative behaviors using requirements goal models
MADAM [Floch et al. 2006]	Facilitating adaptive application development for mobile computing, by representing architecture models at runtime to allow generic middleware components to reason about adaptation
M-Ware [Kumar et al. 2007]	Developing middleware to enable agility, resource-awareness, runtime management and openness in distributed applications, by especially addressing performance concerns and business policies
ML-IDS [Al-Nashif et al. 2008]	Detecting network attacks by inspecting and analyzing the traffic using several levels of granularity (Multi-Level Intrusion Detection System - ML-IDS), and consequently proactively protect the operating system by employing a fusion decision algorithm

Table IV: Comparing Projects in terms of Self-* Properties "√": Supported, "-": Not supported.

	Self-* Properties			
Projects	*Self-Configuring*	*Self-Healing*	*Self-Optimizing*	*Self-Protecting*
Quo	√	-	√	-
IBM Oceano	√	-	√	-
Rainbow	√	√	√	-
Tivoli Risk Manager	-	-	- √	
KX	√	-	-	-
Accord	√	-	-	-
ROC	-	√	-	-
TRAP	√	-	-	-
K-Component	√	-	-	-
Self-Adaptive	√	√	-	-
CASA	√	-	√	-
J3	-	-	√	-
DEAS	√	-	-	-
MADAM	√	-	√	-
M-Ware	√	-	√	-
ML-IDS	-	-	-	√

Table V: Comparing Projects in terms of Adaptation Processes- "H" (High): Provides explicit features to support the process extensively, "M" (Medium): Provides generic features to partially support the process, "L" (Low): Provides limited support, "–" (No Support)- e.g., the vector $(2, 5, 7, 2)/16$ shows there are 2 "no support", 5 "low support", 7 "medium support" and 2 "high support" projects.

	Adaptation Processes			
Projects	Monitoring	Detecting	Deciding	Acting
Quo	L	L	L	M
IBM Oceano	M	M	M	M
Rainbow	H	M	M	M
Tivoli Risk Manager	–	H	–	–
KX	H	M	L	M
Accord	L	L	M	L
ROC	L	L	M	H
TRAP	L	L	–	M
K-Component	L	L	H	L
Self-Adaptive	M	H	H	M
CASA	M	M	L	H
J3	M	M	L	M
DEAS	–	–	M	–
MADAM	M	M	L	M
M-Ware	M	M	M	L
ML-IDS	M	H	M	M
Column-wise Assessment	(2,5,7,2)/16	(1,5,7,3)/16	(2,5,7,2)/16	(2,3,9,2)/16

needs to provide comprehensive solutions to realize all adaptation processes at a high level.

The taxonomy introduced in Section 3 provides a third view. This view is summarized in Table VI, which will also be analyzed column-wise. Before analyzing this table, it should be noted that the possible values for some of the facets are not

mutually exclusive. For example, a project can rely on a hybrid approach including both *making* and *achieving*. The findings related to this view based on each taxonomy facet (corresponding to different columns in the table) are as follows:

—Layer (L): Most of the projects focus on the application layer, which is expected. Note this article deals primarily with the upper layers of a software-intensive system.

—Artifact & Granularity ($A\&G$): Various artifacts at different granularity levels have been addressed, which is a positive point.

—Impact & Cost ($I\&C$): Most of the projects utilize both weak and strong adaptation actions. This is also a positive point because it is possible to use low/high cost and local/global actions depending on the circumstances.

—Making/Achieving (M/A): The achieving approach is rarely observed. This means learning and evolutionary algorithms have not yet been widely used in this area.

—External/Internal (E/I): All of the projects use the external approach, which means that they all support separation of the adaptation mechanism from the application logic.

—Static/Dynamic Decision-Making ($S/D\ DM$): The number of dynamic deciding processes is not too high, but is notable. This is partly due to the research activities in the area of policy-based management.

—Open/Close (O/C): Another remarkable observation is the high number of projects supporting close adaptation. This can be interpreted as the inability to attain openness due to stability and assurance concerns.

—Specific/Generic (S/G): A number of projects (7 out of 16, including J3) have been developed based on component-based systems. The justification is that such components are loosely coupled entities that can be changed dynamically at runtime, easier than the other entities.

—Model-Based/-Free (MB/F): Most of the projects are model-based, which is not surprising noting the wide-spread application of model-based and model-driven approaches in engineering disciplines.

—Reactive/Proactive (R/P): Most of the projects are reactive, which is not generally a disadvantage. However, for some domains, it is required to have proactiveness in order to decrease the aftereffects of changes, or to block change propagation (e.g., faults in safety-critical systems).

—Continuous/Adaptive Monitoring ($C/A\ M$): Most of the projects still use continuous monitoring, which is not preferable noting the cost and the load of this process.

—Human Involvement (HI): Most of the projects do not include an appropriate human interface. This matter impacts the usability and trustworthiness of these systems in practice.

—Interoperability (I): Only one of the projects proposes a mechanism for interoperability with other self-adaptive or autonomic elements or systems. This matter limits their applicability, especially in emerging service-oriented applications.

Table VI: Comparing Projects in terms of the Taxonomy Facets- **"-"**: is not supported, **"?"**: Not clearly known. "L"" Layer, "A&G": Artifact & Granularity, "I&C": Impact & Cost, "M/A": Making/Achieving, "E/I": External/Internal,"S/D DM": Static/Dynamic Decision-Making, "O/C": Open/Close, "S/G": Specific/Generic, "MB/F": Model-Based/-Free, "R/P":Reactive/Proactive, "C/A M": Continuous/Adaptive Monitoring, "HI": Human Involvement, "I":Interoperability.

Taxonomy/ Projects	*Object to adapt*			*Realization*						*Temporal*		*Interaction*	
				Approach			*Type*						
	L	*A&G*	*I&C*	*M/A*	*E/I*	*S/D DM*	*O/C*	*S/G*	*MB/F*	*R/P*	*C/A M*	*HI*	*I*
Quo	application	aspect	w	m	e	s	c	comp.-based	mb	r	c	-	-
IBM Oceano	Infrastructure & network	data center	w/s	m	e	d	o	data center	mb	r	c	can be	-
Rainbow	application	architecture	w/s	m	e	semi-d	c	known arch. styles	mb	r	semi-a	can be	-
Tivoli Risk Manager	network & application	system	-	m	e	-	?	generic	mb	r	c	yes	yes
KX	application	application	w/s	m	e	d	o	generic	mb	r	semi-a	-	-
Accord	application	components	w/s	m	e	d	c	generic	mb	r	c	can be	-
ROC	application & middleware	components/ subsystems	w	m	e	s	c	comp.-based	mb	r	c	-	-
TRAP	application	aspect	w	m	e	-	c	comp.-based	-	r	c	yes	-
K-Component	application	component	w/s	m/a	e	d	c	comp.-based	mb	r/p	c	-	-
Self-Adaptive	application	method	w	m/a	e	semi-d	c	generic	mb	r	c	-	-
CASA	application	component	w/s	m	e	semi-d	c	comp.-based	mb	r	c	-	-
J3	application	aspect	w	m	e	s	c	J2EE app.	mb	r	c	-	-
DEAS	application	application	?	m	e	s	c	generic	-	-	-	-	-
MADAM	middleware	architecture	s	m	e	s	c	comp.-based	mb	r	c	-	-
M-WARE	middleware	parameters	w	m	e	s	o	generic	mb	r	c	-	-
ML-IDS	network	parameters	w	m	e	d	c	generic	mb	r/p	c	-	-

5. RESEARCH CHALLENGES

Self-adaptive software creates new opportunities, and at the same time, poses new challenges to the development and operation of software-intensive systems. This section aims to identify the challenges in realizing self-adaptive software. Prior to classifying the challenges, we first provide a broader view on this issue following the work of Kephart in the context of autonomic computing [Kephart 2005]: i) *Elemen/Component Level Challenges* relate to building element interfaces and contracts to share information, designing/implementing proper adaptation processes, and designing an appropriate architecture for elements in order to execute and coordinate the adaptation processes, ii) *System Level Challenges* relate to coordinating self-* properties and adaptation processes between elements, specifying the evaluation criteria, and defining appropriate architectures to fulfill this level's requirements (e.g., inter-element communication), and iii) *Human-System Interaction Challenges* relate to building trust, providing an appropriate mechanism for collecting user policies, and establishing a proper mechanism to involve humans in the adaptation loop.

Although the above classification provides insight into the challenges associated with self-adaptive systems, it does not quite fit into the taxonomy and landscape discussed in the earlier parts of this article. Moreover, for some of the identified challenges, depending on the underlying design decisions, they may be at the element level or at the system level (e.g., coordinating self-* properties). To remedy this shortcoming, in the following, we aim to classify the underlying challenges based on the points summarized in Fig. 5. This enables us to relate the challenges to the concepts already discussed, and to the analysis performed in the landscape section.

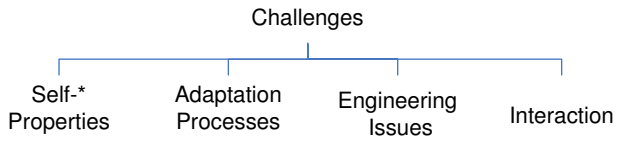

Fig. 5: Classifying Challenges

5.1 Challenges in Engineering Self-Adaptive Software

This section deals with the engineering challenges for requirements analysis, design, implementation, and evaluation of self-adaptive software.

—**Requirements analysis**: As discussed earlier, the system's requirements (especially non-functional), self-* properties, and quality factors are interrelated. Consequently, besides the main task in requirements engineering, that is to capture the stakeholders' expectations, the other key challenges are how to translate, model, and relate those expectations to adaptation requirements and goals to be used at run-time [Kramer and Magee 2007]. The desired model is used as a basis to answer the adaptation requirements' questions (described in Section 2.3) in the *developing* and *operating* phases. In other words, the specifications are not

only used in developing self-adaptive software, but also kept for conflict resolution and change management at the operating phase. Goal-oriented requirements engineering appears to be a promising approach for addressing these challenges.

—**Design issues**: One key challenge is how to design self-adaptive software to fulfill the adaptation requirements. Although numerous research efforts have focused on designing self-adaptive software, there are still some major open problems in this area. In an internal approach, one needs to extend the existing programming languages, or define new adaptation languages. However, as the projects' landscape in Section 4.2 indicates, most existing research is based on external approaches. Consequently, the challenge is decomposed into: i) designing the underlying adaptable software system(s) and adaptation engine(s), and ii) forming the architecture connecting them together. The following list provides a summary of these challenges.

The first question is how to design adaptable software, either from scratch or from a legacy system. This question deals with the important issue of *design for adaptability*. Adaptable software needs to expose important information through sensors (by events), and to facilitate effectors for variability. The related outstanding questions are as follows: i) Which architecture styles and design patterns are appropriate for this purpose? ii) Which component model provides the best support for the sensing and effecting in vivo mechanisms? iii) Which interfaces and contracts need to be considered? And most importantly, iv) How to use re-engineering approaches to transform a legacy system into an adaptable software? Another challenge is how to use existing experiences in re-engineering, reverse engineering, and re-factoring, as discussed in [Muller 2006], in dealing with Ultra-Large-Scale (ULS) systems [Northrop 2006]. Parekh *et al.* address this challenge partially by proposing a framework for retrofitting autonomic capabilities into legacy systems [Parekh et al. 2006]. It appears that for variability management, ideas from product line architecture would be useful as well.

The primary challenges for *designing the adaptation engine* will be addressed in Section 5.3 for each adaptation process. However, there are two missing points: i) How to design the interfaces required for policy enforcement and connection to adaptable software? ii) What kind of architecture is suitable for inter-process communication and sharing knowledge inside the engine?

Last, but not least, is the issue of interoperability and system-wide architecture. Several researchers have investigated this issue. Smith *et al.* [Smith et al. 2005] address complexity and changeability of interaction between system elements, or system of systems. McKinley *et al.* discuss the need for coordinating components across layers of an adaptive system [McKinley et al. 2004]. One aspect of this challenge is the availability of communication channels between elements, which is addressed in [Parekh et al. 2006].

—**Implementation languages, tools, and frameworks**: Generally speaking, building self-adaptive and autonomic software can be accomplished in two ways [Liu et al. 2004]: i) extending existing programming languages/systems or defining new adaptation languages (e.g., based on aspects [Duzan et al. 2004]), and ii) enabling dynamic adaptation by allowing adding, removing, and modifying software entities at run-time (e.g., based on middleware [Floch et al. 2006; Asadollahi

et al. 2009]). In practice, a combination of these solutions is required to build elements, to facilitate their compositions, and to provide a run-time infrastructure for interaction and management. Although numerous research efforts have investigated both solutions and their combination, there is still a lack of powerful languages, tools, and frameworks that could help realize adaptation processes and instrument sensors/effectors in a systematic manner. The middleware-based approach appears more promising, since the infrastructure is available. However, dynamic adaptation actions (i.e. effectors) are still not extensively supported, or not reliable.

—**Testing and assurance**: Testing and assurance are probably the least focused phases in engineering self-adaptive software, and there are only a few research efforts addressing this topic. The challenging issue is the availability of several alternatives for adaptable artifacts and parameters in the system. This leads to several paths of execution in different scenarios. If we add the dynamic decision-making and achieving approach to this scenario, it will become even more complex. There are attempts to use a self-test mechanism at runtime to validate the changes. King *et al.* present this idea for autonomic systems by adding a test manager for this purpose [King et al. 2007]. Zhang also proposes a run-time model checking approach for the verification of adaptation [Zhang 2007].

—**Evaluation and quality of adaptation**: So far there has not been any comprehensive work addressing evaluation criteria or metrics for self-adaptive software, or more generally, for autonomic computing. There are links between self-* properties and software quality goals; however, how quality metrics can help measure the quality of adaptation is still an open question. Factors like safety, security, and performance are discussed in [Gjorven et al. 2006], and QoS, cost, and granularity/flexibility are discussed in [McCann and Huebscher 2004]. The availability of benchmarks, testbeds, and appropriate case studies can help in evaluating and comparing different adaptation solutions, at least in relation to each adaptation process.

5.2 Challenges Related to Self-* Properties

Self-* properties are the key features of self-adaptive software. This section summarizes the challenges expected in realizing these properties, both individually and collectively.

—**Individual self-* properties**: According to the analysis performed in Section 4.2, self-protecting has received the least attention among the self-* properties. Generally, most of research on self-protecting (not necessarily in the scope of this paper) focus on detecting anomaly symptoms (e.g., intrusion). Some of this research also concentrates on integrating various technologies for security management and recovery (e.g., [Tuttle et al. 2003]). However, the point is that realizing all of the adaptation processes for this property, particularly at the upper layers (e.g., middleware and application), is still quite challenging.
An important question in realizing self-* properties is how well the system is capable of detecting changes and their potential consequences in the adaptable software or its context. These challenges include inferring or predicting the change propagation based on the dynamic model of software or data analysis at run-time.

Also, specific to the self-healing and self-protecting properties, the problem is how to limit or isolate the problematic components, and eventually how to recover these components. This matter definitely needs implanted effectors that allow recovery without a crash or interruption in the system.

—**Building multi-property self-adaptive software**: As shown in Table VI, the majority of projects do not address more than one self-* property. Moreover, those projects that adress multiple properties, do not systematically coordinate them. Generally, most of the proposed solutions do not address the relationships between self-* properties including priority, conflict, and the execution order of their actions at run-time. It is clear that coordinating and orchestrating these properties and their derived goals at different levels of granularity is one of the significant challenges in self-adaptive software. IBM addresses this issue in its reference architecture for orchestrating across and within disciplines for self-* properties [IBM Ref. Arch. 2005]. Cheng *et al.* also focus on this challenge in an architecture-based adaptation [Cheng et al. 2006].
It is also important to note that each self-* property deals with several concerns, such as cost and time. Realizing desired self-* properties means satisfying certain goals related to these concerns, subject to given constraints. The problem with the available solutions is that they usually do not rely on a multi-concern view in the adaptation. An example of such a missing concern is the cost/benefit of actions related to the business aspects of a software system.

5.3 Challenges in Adaptation Processes

An appropriate way to study challenges is to classify them on the basis of adaptation processes:

—**Monitoring challenges**: A significant challenge for monitoring different attributes in adaptable software is the cost/load of the sensors. In most cases, a number of in vivo methods collect various information, which may not be required by the desired self-* properties. In some cases, the monitoring process does not need the details of the events, while in the case of deviating from "normal" behavior, more data will be required. Consequently, a monitoring process needs to be adapted regarding the adaptable software situation, in order to increase the level of awareness. Such a process can be called an *adaptive monitoring* process. Oreizy *et al.* argue that a self-adaptive software needs a planning process to specify which observations are necessary to decide "when" and "where" adaptations are required [Oreizy et al. 1999]. A few efforts address adaptive monitoring, such as the COMPAS framework in J2EE applications [Diaconescu et al. 2004]. Although these efforts have partially addressed the monitoring challenges, this subject deserves much more attention.

—**Detecting challenges**: The prominent question in the detecting process is "which behaviors/states of a software system are healthy/normal?". Answering this question often requires a time-consuming static and dynamic analysis of the system, which may also be strongly affected by the underlying random variables (i.e., users' requests arrival times, and faults in different components). Although there have been efforts to apply statistical and data mining techniques to address this issue (e.g., for problem determination [Brown et al. 2001]), the

existing realizations of this process are still mostly ad hoc and incomplete.

—**Deciding challenges**: The deciding process still needs lots of attention both at the local level (adaptation engine) and at the system level. As discussed in [McKinley et al. 2004], most of the known approaches are static and only effective in specific domains. As shown in Table V, few research projects have provided strong support for this process, in particular by focusing on multiple self-* properties in dynamic and uncertain environments. Moreover, as shown in Table VI, about half of the projects address dynamic decision-making. In the presence of multiple objectives, in addition to the necessity of deciding online and dynamically, one faces the following additional challenges: i) finding approximately or partially optimal solutions for multi-objective decision-making problems, ii) dealing with uncertainty and incompleteness of events/information from the system's *self* and *context*, iii) correlating local and global decision-making mechanisms, and iv) addressing the scalability and fault-proneness of the decision-making mechanism using centralized or decentralized models.

—**Acting challenges**: One important challenge is how to assure that the adaptation is going to be stable and have a predictable impact on the functional and non-functional aspects of the underlying software system. It is important to know: i) whether the adaptation actions follow the contracts and the architectural styles of the system, ii) whether they impact the safety/integrity of the application, and iii) what will happen if the action fails to complete, or if preemption is required in order to suspend the current action and deal with a higher priority action. These issues are critical, particularly for systems with open adaptation and dynamic decision-making in non-stationary environments. These challenges have been addressed by Laddaga [Laddaga 2006] in the context of stability of self-adaptive software, and have been linked to verification by Karsai *et al.* [Karsai et al. 2001]. These issues still require much more research, since most of the solutions presented in the literature are ad hoc and problem-specific. Formal methods and model-driven solutions, with the aid of model/constraint checking, seem to be a promising direction in this respect.

5.4 Challenges in Interaction

At first glance, a human interface for self-adaptive software appears to be much easier to build compared to non-adaptive software. However, as Russel *et al.* [Russell et al. 2003] point out, several problems exist that include policy management, trust, and human involvement. The analysis in Section 4.2 also shows that most of the projects do not have a human in the loop for policy changing or tracing adaptation processes. These challenges can be briefly described as follows.

—**Policy management**: One prominent downside of some of the existing solutions is the lack of explicit representation of policies and goals. This leads to two problems that will be explained next.

- Policy translation: The policies and goals often need to be decomposed or translated into lower-level/local ones that are understandable by the system elements. Without having a goal/policy model, it is difficult to accomplish this task effectively and efficiently in complex large-scale systems. This matter needs highly flexible models and algorithms, which is a research issue definitely

worth investigating.

- Dynamic policies and goals: The developers need to hard-code or precompile the action selection mechanism for the deciding process. A rule-based mechanism based on a fixed static conflict resolution mechanism is commonly used for this purpose. The rules in such systems are hand-coded or compiled on the basis of declarative descriptions of objectives, policies, or desired behaviors in the design phase. However, the goals and management policies may be subject to change during the operating phase.

—**Building trust**: Another noteworthy challenge issue is how to establish trust. The issue of trust is not limited to self-adaptive software and is a general research topic in many computer-based and software-intensive systems. However, self-adaptive software, due to its dynamic and automatic nature, adds new concerns to this problem. The independence and intelligence may make this type of system less traceable for users and stakeholders. It is essential that a self-adaptive application facilitates trust management for the security concerns, and also reports its activities and decisions to administrators in order to expose what is going on. Trust can be built incrementally to ensure that the adaptation processes are safe and secure. It is noteworthy that trust can also be defined between self-adaptive elements and services, in which this issue will affect interoperability.

—**Interoperability**: This issue is challenging in most distributed complex systems and particularly in the so-called "systems of systems". In self-adaptive software, besides the data-related concerns, coordinating and orchestrating self-adaptation behavior of all elements is a challenging task. Fulfilling global requirements and self-* properties, for each property and across different properties, is not a straight-forward task. The emergence of Ultra-Large Scale (ULS) systems adds to the significance of interoperability, and at the same time, creates new challenges in this regard [Northrop 2006].

6. SUMMARY

The area of self-adaptive software enjoys a growing importance. In spite of numerous excellent research efforts, this area is still in its infancy, and the existing body of knowledge is far from being adequate to address the escalating demands for self-adaptivity of software in today's dynamic and ever-changing environments. Self-adaptive software poses many new opportunities, as well as challenges, for computer scientists and engineers. New models and theories are needed to access to these opportunities and to cope with the associated challenges towards fulfilling the requirements.

This survey article has discussed the basic principles behind self-adaptive software and proposed a taxonomy of adaptation. The questions of *where*, *when*, *what*, *why*, *who*, and *how* form the basis of this taxonomy. A landscape has been presented based on reviewing a number of disciplines related to self-adaptive software, as well as some selected research projects. A comparison between the different views of this landscape has provided a framework to identify gaps. For example, the self-protecting property needs more work to provide more secure software systems. Various threats to on-line distributed systems are the driving forces for this issue. Adaptation processes also need to be improved to adapt software systems effectively

and efficiently.

The landscape also helps locate future challenges in this emerging research area. Such challenges have been classified into four categories, namely self-* properties, adaptation processes, engineering issues, and interaction. These categories are based on the discussed fundamentals of self-adaptive software in the previous sections. The challenges have been linked to related discussions and concepts.

REFERENCES

Abdelwahed, S., Kandasamy, N., and Neema, S. 2004. A control-based framework for self-managing distributed computing systems. In *Proc. of Workshop on Self-healing Sys.* 3–7.

Al-Nashif, Y., Kumar, A., Hariri, S., Luo, Y., Szidarovsky, F., and Qu, G. 2008. Multi-Level Intrusion Detection System (ML-IDS). In *Autonomic Computing, 2008. ICAC'08. International Conference on*. 131–140.

Alur, D., Crupi, J., and Malks, D. 2001. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice-Hall.

Aly, W. H. F. and Lutfiyya, H. 2007. Dynamic adaptation of policies in data center management. In *Proc. of IEEE Int. Workshop on Policies for Distributed Sys. & Networks*. 266–272.

Amoui, M., Salehie, M., Mirarab, S., and Tahvildari, L. 2008. Adaptive action selection in autonomic software using reinforcement learning. In *Proc. of Int. Conf. on Autonomic and Autonomous Systems*. 175–181.

Appleby, K., Fakhouri, S., Fong, L., Goldszmidt, G., Kalantar, M., Krishnakumar, S., Pazel, D., Pershing, J., and Rochwerger, B. 2001. Oceano - SLA-based management of a computing utility. In *Proc. of IFIP/IEEE Int. Symp. on Integrated Network Management*. 855–868.

ARM. Application response measurement. http://www.opengroup.org/tech/management/arm/.

Arshad, N., Heimbigner, D., and Wolf, A. 2003. Deployment and dynamic reconfiguration planning for distributed software systems. In *Proc. of IEEE Conf. on Tools with Artificial Intelligence*. 39–46.

Asadollahi, R., Salehie, M., and Tahvildari, L. 2009. Starmx: A framework for developing self-managing java-based systems. In *Proc of ICSE workshop on Software Engineering for Adaptive and Self-Managing Systems*. TBA.

Babaoglu, O., Canright, G., Deutsch, A., Caro, G. A. D., Ducatelle, F., Gambardella, L. M., Ganguly, N., Jelasity, M., Montemanni, R., Montresor, A., and Urnes, T. 2006. Design patterns from biology for distributed computing. *ACM Trans. on Autonomous and Adaptive Sys. 1,* 1, 26–66.

Babaoglu, O., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A., and van Steen, M. 2005. *Self-star Properties in Complex Information Systems: Conceptual and Practical Foundations (Lecture Notes in Computer Science)*.

Badr, N., Taleb-Bendiab, A., and Reilly, D. 2004. Policy-based autonomic control service. In *Proc. of IEEE Int. Workshop on Policies for Distributed Sys. & Networks*. 99.

Bantz, D. F., Bisdikian, C., Challener, D., Karidis, J. P., Mastrianni, S., Mohindra, A., Shea, D. G., and Vanover, M. 2003. Autonomic personal computing. *IBM Systems Journal 42,* 1, 165–176.

Bernon, C., Gleizes, M., Peyruqueou, S., and Picard, G. 2003. Adelfe: A methodology for adaptive multi-agent systems engineering. *Lecture Notes in Computer Science*, 156–169.

Bhat, V., Parashar, M., Liu, H., Khandekar, M., Kandasamy, N., and Abdelwahed, S. 2006. Enabling self-managing applications using model-based online control strategies. In *Proc. of IEEE Int. Conf. on Autonomic Computing*. 15–24.

Bigus, J. P., Schlosnagle, D. A., Pilgrim, J. R., Mills, W. N., and Diao, Y. 2002. Able: A toolkit for building multiagent autonomic systems. *IBM Sys. Journal 41,* 3, 350–371.

Birman, K., van Renesse, R., and Vogels, W. 2004. Adding high availability and autonomic behavior to web services. In *Proc. of Int. Conf. on Software Eng.* 17–26.

Boutilier, C., Das, R., Kephart, J. O., and Walsh, W. E. 2003. Towards cooperative negotiation for decentralized resource allocation in autonomic computing systems. In *Proc. of Int. Joint Conf. on Artificial Intelligence*. 1458–1459.

Bradbury, J. S., Cordy, J. R., Dingel, J., and Wermelinger, M. 2004. A survey of self-management in dynamic software architecture specifications. In *Proc. of ACM workshop on Self-managed systems*. 28–33.

Brown, A., Kar, G., and Keller, A. 2001. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Proc. of Integrated Network Management*. 377–390.

Buckley, J., Mens, T., Zenger, M., Rashid, A., and Kniesel, G. 2005. Towards a taxonomy of software change. *Journal on Software Maintenance and Evolution: Research and Practice*, 309–332.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. 1996. *A system of patterns - pattern oriented software architecture*. Wiley.

Candea, G., Cutler, J., and Fox, A. 2004. Improving availability with recursive microreboots: a soft-state system case study. *Performance Evaluation 56,* 1-4, 213–248.

Candea, G., Kiciman, E., Kawamoto, S., and Fox, A. 2006. Autonomous recovery in componentized internet applications. *Cluster Computing 9,* 1, 175–190.

Cardellini, V., Colajanni, M., and Yu, P. 1999. Dynamic load balancing on Web-server systems. *IEEE Internet Computing 3,* 3, 28–39.

Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. 2001. Design and evaluation of a wide-area event notification service. *ACM Trans. on Computer Sys. 19,* 3, 332–383.

Cheng, S.-W., Garlan, D., and Schmerl, B. 2006. Architecture-based self-adaptation in the presence of multiple objectives. In *Proc. of Workshop on Self-adaptation and Self-managing Sys.* 2–8.

CIM. Common information model standard. http://www.dmtf.org/standards/cim/.

de Lemos, R. and Fiadeiro, J. L. 2002. An architectural support for self-adaptive software for treating faults. In *Proc. of Workshop on Self-healing Sys.* 39–42.

Diaconescu, A., Mos, A., and Murphy, J. 2004. Automatic performance management in component based software systems. In *Proc. of Int. Conf. on Autonomic Computing*. 214–221.

Dilman, M. and Raz, D. 2002. Efficient reactive monitoring. *IEEE Journal on Selected Areas in Communications 20,* 4, 668–676.

Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., and Zambonelli, F. 2006. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS) 1,* 2, 223–259.

Dowling, J. 2004. The decentralised coordination of self-adaptive components for autonomic distributed systems. Ph.D. thesis, Department of Computer Science, Trinity College Dublin.

Dowling, J. and Cahill, V. 2004. Self-managed decentralised systems using K-components and collaborative reinforcement learning. In *Proc. of ACM Workshop on Self-Managed Sys.* 39–43.

Duzan, G., Loyall, J. P., Schantz, R. E., Shapiro, R., and Zinky, J. A. 2004. Building adaptive distributed applications with middleware and aspects. In *Proc. of Int. Conf. on Aspect-Oriented Software Development*. 66–73.

Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., and Gjørven, E. 2006. Using Architecture Models for Runtime Adaptability. *IEEE SOFTWARE*, 62–70.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley.

Ganek, A. G. and Corbi, T. A. 2003. The dawning of the autonomic computing era. *IBM Sys. Journal, Special Issues on Autonomic Computing 42*, 5–18.

Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., and Steenkiste, P. 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer 37,* 10, 46–54.

Garlan, D. and Schmerl, B. 2002. Model-based adaptation for self-healing systems. In *Proc. of Workshop on Self-healing Sys.* 27–32.

Georgas, J. C., van der Hoek, A., and Taylor, R. N. 2005. Architectural runtime configuration management in support of dependable self-adaptive software. In *WADS '05: Proceedings of the 2005 workshop on Architecting dependable systems*. 1–6.

Gjorven, E., Eliassen, F., and Aagedal, J. O. 2006. Quality of adaptation. In *Proc. of Int. Conf. on Autonomic and Autonomous Sys.* 9–14.

Greenwood, P. and Blair, L. 2004. Using dynamic aspect-oriented programming to implement an autonomic system. In *Proc. of Dynamic Aspects Workshop*. 76–88.

Hinchey, M. G. and Sterritt, R. 2006. Self-managing software. *IEEE Computer 39,* 2, 107–109.

Horn, P. 2001. Autonomic computing: IBM's perspective on the state of information technology. http://www-1.ibm.com/industries/government/doc/content/bin/auto.pdf.

Howe, A. E. 1995. Improving the reliability of artificial intelligence planning systems by analyzing their failure recovery. *IEEE Trans. on Knowledge and Data Eng. 7,* 1, 14–25.

Huebscher, M. C. and McCann, J. A. 2008. A survey of autonomic computing—degrees, models, and applications. *ACM Comput. Surv. 40,* 3, 1–28.

Hutchison, D., Coulson, G., Campbell, A., and Blair, G. S. 1994. Quality of service management in distributed systems. 273–302.

IBM. 2005. Autonomic computing toolkit: Developers guide. Tech. Rep. SC30-4083-03.

IBM-AC 2001. Autonomic computing 8 elements. http://www.research.ibm.com/autonomic/overview/elements.html.

IBM BtM. Eclipse BtM (Build to Manage). www.ibm.com/developerworks/eclipse/btm.

IBM Ref. Arch. 2005. An architectural blueprint for autonomic computing. IBM white paper. http://www-03.ibm.com/autonomic/pdfs/AC Blueprint White Paper V7.pdf.

IBM SMART. SMART. http://www.almaden.ibm.com/software/dm/SMART/.

IEEE-ISO/IEC 14764 2006. Standard for software maintenance - IEEE 14764-2006 - ISO/IEC 14764. URL = http://ieeexplore.ieee.org/iel5/11168/35960/01703974.pdf.

ISO/IEC 9126-1 2001. ISO/IEC 9126-1 Standard: Software Eng. -Product quality - Part 1: Quality model, Int. Standard Organization, 2001.

Jelasity, M., Babaoglu, O., Laddaga, R., Nagpal, R., Zambonelli, F., Sirer, E. G., Chaouchi, H., and Smirnov, M. 2006. Interdisciplinary research: Roles for self-organization. *IEEE Intelligent Sys. 21,* 2, 50–58.

JMX. Sun Java Management eXtensions. http://jcp.org/en/jsr/detail?id=3.

JVMTI. Sun JVM Tool Interface. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/.

Kaiser, G. E., Parekh, J., Gross, P., and Valetto, G. 2003. Kinesthetics extreme: An external infrastructure for monitoring distributed legacy systems. In *Proc. of Active Middleware Services*. 22–31.

Karamanolis, C., Karlsson, M., and Zhu, X. 2005. Designing controllable computer systems. In *Proc. of Hot Topics in Operating Sys. workshop*. 49–54.

Karsai, G., Lédeczi, Á., Sztipanovits, J., Péceli, G., Simon, G., and Kovácsházy, T. 2001. An approach to self-adaptive software based on supervisory control. In *Proc. of Int. Workshop on Self-Adaptive Software*. 24–38.

Karsai, G. and Sztipanovits, J. 1999. A model-based approach to self-adaptive software. *IEEE Intelligent Systems 14,* 3, 46–53.

Keeney, J. and Cahill, V. 2003. Chisel: A policy-driven, context-aware, dynamic adaptation framework. In *Proc. of IEEE Int. Workshop on Policies for Distributed Sys. & Networks*. 3–14.

Keeney, R. L. and Raiffa, H. 1976. *Decisions with Multiple Objectives*. Wiley.

Kephart, J. O. 2005. Research challenges of autonomic computing. In *Proc. of Int. Conf. on Software Eng.* 15–22.

Kephart, J. O. and Chess, D. M. 2003. The vision of autonomic computing. *IEEE Computer 36,* 1, 41–50.

Kephart, J. O. and Walsh, W. 2004. An artificial intelligence perspective on autonomic computing policies. In *Proc. of IEEE int. workshop on Policies for Dist. Sys. and Networks*. 3–13.

King, T. M., Babich, D., Alava, J., Clarke, P. J., and Stevens, R. 2007. Towards self-testing in autonomic computing systems. In *Proc. of the Eighth Int. Symp. on Autonomous Decentralized Systems*. 51–58.

Kokar, M. M., Baclawski, K., and Eracar, Y. A. 1999. Control theory-based foundations of self-controlling software. *IEEE Intelligent Sys. 14,* 3, 37–45.

Kon, F., Costa, F., Blair, G., and Campbell, R. 2002. The case for reflective middleware. *Communications of the ACM 45,* 6, 33–38.

Kramer, J. and Magee, J. 1990. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on software engineering 16,* 11, 1293–1306.

Kramer, J. and Magee, J. 2007. Self-managed systems: an architectural challenge. In *Proc. of ICSE - Future of Software Engineering*. 259–268.

Kumar, V., Cooper, B., Cai, Z., Eisenhauer, G., and Schwan, K. 2007. Middleware for enterprise scale data stream management using utility-driven self-adaptive information flows. *Cluster Computing 10,* 4, 443–455.

Laddaga, R. 1997. Self-adaptive software. Tech. Rep. 98-12, DARPA BAA.

Laddaga, R. 1999. Guest editor's introduction: Creating robust software through self-adaptation. *IEEE Intelligent Sys. 14,* 3, 26–29.

Laddaga, R. 2000. Active software. In *Proc. of Int. Workshop on Self-Adaptive Software*. 11–26.

Laddaga, R. 2006. Self adaptive software problems and projects. In *Proc. of IEEE workshop on Software Evolvability*. 3–10.

Laddaga, R., Robertson, P., and Shrobe, H. 2000. Introduction to self-adaptive software: Applications. In *Proc. of Int. Workshop on Self-Adaptive Software*. Vol. 2614. 1–5.

Laddaga, R., Robertson, P., and Shrobe, H. E. 2001. Results of the 2^{nd} Int. workshop on self-adaptive software. In *Proc. of Int. Workshop on Self-Adaptive Software*. 281–290.

Landauer, C. and Bellman, K. L. 2001. New architectures for constructed complex systems. *Applied Mathematics and Computation 120*, 149–163.

Lapouchnian, A., Liaskos, S., Mylopoulos, J., and Yu, Y. 2005. Towards requirements-driven autonomic systems design. In *Proc. of Workshop on Design and Evolution of Autonomic App. Software*. 1–7.

Lehman, M. M. 1996. Laws of software evolution revisited. In *Proc. of European Workshop on Software Process Technology*. 108–124.

Lieberherr, K. J. and Palsberg, J. 1993. Engineering adaptive software. Projest Proposal, ftp://ftp.ccs.neu.edu/pub/people/lieber/proposal.ps.

Litoiu, M., Woodside, M., and Zheng, T. 2005. Hierarchical model-based autonomic control of software systems. In *Proc. of Workshop on Design and Evolution of Autonomic App. Software*. 27–33.

Liu, H., Parashar, M., and Hariri, S. 2004. A component-based programming model for autonomic applications. In *Proc. of Int. Conf. on Autonomic Computing*. 10–17.

Loyall, J. P., Bakken, D. E., Schantz, R. E., Zinky, J. A., Karr, D. A., Vanegas, R., and Anderson, K. R. 1998. QoS aspect languages and their runtime integration. In *Proc. of Int. Workshop on Languages, Compilers, and Run-Time sys. for scalable computers*. 303–318.

Lutfiyya, H., Molenkamp, G., Katchabaw, M., and Bauer, M. A. 2001. Issues in managing soft QoS requirements in distributed systems using a policy-based framework. In *Proc. of IEEE Int. Workshop on Policies for Distributed Sys. & Networks*. 185–201.

Maes, P. 1990. Situated agents can have goals. *Robotics and Autonomous Sys. 6*, 49–70.

Magee, J. and Kramer, J. 1996. Dynamic structure in software architectures. *ACM SIGSOFT Software Engineering Notes 21,* 6, 3–14.

McCann, J. A. and Huebscher, M. C. 2004. Evaluation issues in autonomic computing. In *Grid and Cooperative Computing workshops*. 597–608.

McCann, J. A., Lemos, R. D., Huebscher, M., Rana, O. F., and Wombacher, A. 2006. Can self-managed systems be trusted? some views and trends. *Knowledge Eng. Review 21,* 3, 239–248.

McKinley, P. K., Sadjadi, M., Kasten, E. P., and Cheng, B. H. C. 2004. Composing adaptive software. *IEEE Computer*, 56–64.

Menascé, D. A. and Bennani, M. N. 2006. Autonomic virtualized environments. In *Proc. of Int. Conf. on Autonomic and Autonomous Systems*. 28.

M.H. Klein, R. Kazman, L. B. J. C. M. B. and Lipson, H. 1999. Attribute-based architectural styles. In *Proc. of the IEEE/IFIP First Workshop Conf. on Software Architecture*. 225–243.

Morandini, M., Penserini, L., and Perini, A. 2008. Towards goal-oriented development of self-adaptive systems. In *Proc. of Int. workshop on Software Engineering for Adaptive and Self-managing Systems*. 9–16.

Mukhija, A. and Glinz, M. 2005. Runtime adaptation of applications through dynamic recomposition of components. In *Proc. of Int. Conf. on Architecture of Computing Sys.* 124–138.

Muller, H. A. 2006. Bits of history, challenges for the future and autonomic computing technology. In *Proc. of Working Conf. on Reverese Eng.* 9–15.

Murch, R. 2004. *Autonomic Computing*. Prentice Hall.

Northrop, L. 2006. Ultra-large-scale systems: The software challenges of the future. Tech. rep., Carnegie Mellon University. July. http://www.sei.cmu.edu/uls/.

Nowicki, T., Squillante, M. S., and Wu, C. W. 2005. Fundamentals of dynamic decentralized optimization in autonomic computing systems. In *LNCS*. Vol. 3460. 204–218.

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

Oreizy, P., Medvidovic, N., and Taylor, R. N. 1998. Architecture-based runtime software evolution. In *Proc. of Int. Conf. on Software Eng.* 177–186.

Parashar, M. and Hariri, S. 2005. Autonomic computing: An overview. *Hot Topics, Lecture Notes in Computer Science 3566*, 247–259.

Parekh, J., Kaiser, G., Gross, P., and Valetto, G. 2006. Retrofitting autonomic capabilities onto legacy systems. *Cluster Computing 9,* 2, 141–159.

Parnas, D. L. 1985. Software aspects of strategic defense systems. *Commun. ACM 28,* 12, 1326–1335.

Patrascu, R., Boutilier, C., Das, R., Kephart, J. O., Tesauro, G., and Walsh, W. E. 2005. New approaches to optimization and utility elicitation in autonomic computing. In *Proc. of Conf. on Artificial Intelligence*. 140–145.

Pavlovic, D. 2000. Towards semantics of self-adaptive software. In *Proc. of Int. Workshop on Self-Adaptive Software*. Lecture Notes in Computer Science, vol. 1936. 65–74.

Pawlak, R., Seinturier, L., Duchien, L., and Florin, G. 2001. JAC: A flexible solution for aspect-oriented programming in Java. In *Proc. of Metalevel Architectures and Separation of Crosscutting Concerns*. 1–24.

Peltz, C. 2003. Web services orchestration and choreography. *IEEE Computer 36,* 10, 46–52.

Pinto, M., Fuentes, L., Fayad, M., and Troya, J. 2002. Separation of coordination in a dynamic aspect oriented framework. In *Proc. of Int. Conf. on Aspect-Oriented Software Development*. 134–140.

Poladian, V., Sousa, J. P., Garlan, D., and Shaw, M. 2004. Dynamic configuration of resource-aware services. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 604–613.

Popovici, A., Gross, T., and Alonso, G. 2002. Dynamic weaving for aspect-oriented programming. In *Proc. of Int. Conf. on Aspect-Oriented Software Development*. 141–147.

Porcarelli, S., Castaldi, M., Giandomenico, F. D., Bondavalli, A., and Inverardi, P. 2003. A framework for reconfiguration-based fault-tolerance in distributed systems. In *Proc. of ICSE Workshop on Architecting Dependable Systems II*. Lecture Notes in Computer Science. 167–190.

Qu, G. and Hariri, S. 2007. *Autonomic computing: concepts, infrastructures, and applications*. CRC, Chapter Anomaly-based self-protection against network attacks, 493–521.

Ramadge, P. J. and Wonham, W. M. 1987. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization 25,* 1, 206–230.

Robertson, P. and Laddaga, R. 2005. Model based diagnosis and contexts in self adaptive software. In *Proc. of Self-* Properties in Complex Information Sys.* 112–127.

Robertson, P. and Williams, B. 2006. Automatic recovery from software failure. *Commun. ACM 49,* 3, 41–47.

Russell, D. M., Maglio, P., Dordick, R., and Neti, C. 2003. Dealing with ghosts: Managing the user experience of autonomic computing. *IBM Sys. Journal 42,* 1, 177–188.

Russell, S. J. and Norvig, P. 1995. *Artificial intelligence: a modern approach.* Prentice-Hall.

Saab, C. B., Bonnaire, X., and Folliot, B. 2002. Phoenix: A self adaptable monitoring platform for cluster management. *Cluster Computing 5,* 1, 75–85.

Sadjadi, S. M. and McKinley, P. K. 2004. ACT: An adaptive CORBA template to support unanticipated adaptation. In *Proc. of Int. Conf. on Dist. Computing Sys.* 74–83.

Sadjadi, S. M., McKinley, P. K., Cheng, B. H. C., and Stirewalt, R. E. K. 2004. TRAP/J: Transparent generation of adaptable Java programs. *Lecture Notes in Computer Science 3291*, 1243–1261.

Salehie, M., Li, S., Asadollahi, R., and Tahvildari, L. 2009. Change support in adaptive software: A case study for fine-grained adaptation. In *Proc. of IEEE Conf. and Workshops on Engineering of Autonomic and Autonomous Systems.* TBA.

Salehie, M., Li, S., and Ladan, T. 2009. Employing aspect composition in adaptive software systems: A case study. In *Proc. of ACM Practices of Linking Aspect Technology and Evolution workshop.* TBA.

Salehie, M. and Tahvildari, L. 2005a. Autonomic computing: emerging trends and open problems. In *Proc. of Workshop on Design and Evolution of Autonomic App. Software.* 82–88.

Salehie, M. and Tahvildari, L. 2005b. A policy-based decision making approach for orchestrating autonomic elements. In *Proc. of IEEE Int. Workshop on Software Tech. & Eng. Prac.* 173–181.

Salehie, M. and Tahvildari, L. 2007. A weighted voting mechanism for action selection problem in self-adaptive software. In *Proc. IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems.* 328–331.

Schmidt, D. C. 2002. Middleware for real-time and embedded systems. *Communication of ACM 45,* 6, 43–48.

Schmidt, D. C. and Cleeland, C. 1999. Applying patterns to develop extensible orb middleware. *IEEE Communications Magazine 37*, 54–63.

Scott, J., Neema, S., Bapty, T., and Abbott, B. 2000. Hardware/software runtime environment for dynamically reconfigurable systems. Tech. Rep. ISIS-2000-06, Vanderbilt Univ.

Serugendo, G. D. M., Foukia, N., Hassas, S., Karageorgos, A., Mostéfaoui, S. K., Rana, O. F., Ulieru, M., Valckenaers, P., and van Aart, C. 2003. Self-organisation: Paradigms and app. In *Proc. of Eng. Self-Organising App. workshop.* 1–19.

ShiftOne JRat. JRat (Java Runtime Analysis Toolkit). http://jrat.sourceforge.net/.

Sloman, M. 1994. Policy driven management for distributed systems. *J. Network Syst. Manage. 2,* 4.

Smith, D., Morris, E., and Carney, D. 2005. Interoperability issues affecting autonomic computing. In *Proc. of Workshop on Design and Evolution of Autonomic App. Software.* 89–91.

SNMP. Simple network management protocol. http://www.ietf.org/html.charters/OLD/snmpcharter.html.

Srivastava, B. and Kambhampati, S. 2005. The case for automated planning in autonomic computing. In *Proc. of Int. Conf. on Automatic Computing.* 331–332.

Sterritt, R. 2003. Autonomic computing: the natural fusion of soft computing and hard computing. In *IEEE Int. Conf. on Sys., Man and Cybernetics.* Vol. 5. 4754–4759.

Sterritt, R. and Bustard, D. W. 2003. Autonomic computing - a means of achieving dependability? In *Proc. of IEEE Symp. and Workshops on Eng. of Computer-Based Sys.* 247–251.

Sterritt, R., Parashar, M., Tianfield, H., and Unland, R. 2005. A concise introduction to autonomic computing. *Advanced Eng. Informatics 19*, 181–187.

Subramanian, N. and Chung, L. 2001. Software architecture adaptability: An nfr approach. In *Proc. of Int. Workshop on Principles of Software Evolution*. 52–61.

Suvée, D., Vanderperren, W., and Jonckers, V. 2003. JAsCo: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*. 21–29.

Tanenbaum, A. S. and Woodhull, A. S. 2006. *Operating systems: design and implementation*, Third ed. Pearson Prentice Hall.

Tesauro, G. 2007. Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Computing 11,* 1, 22–30.

Tesauro, G., Chess, D. M., Walsh, W. E., Das, R., Segal, A., Whalley, I., Kephart, J. O., and White, S. R. 2004. A multi-agent systems approach to autonomic computing. In *Proc. of Int. Conf. on Autonomous Agents and Multiagent Systems*. 464–471.

Tuttle, S., Batchellor, V., Hansen, M. B., and Sethuraman, M. 2003. Centralized risk management using tivoli risk manager 4.2. Tech. rep., IBM Tivoli Software. December.

Tziallas, G. and Theodoulidis, B. 2004. A controller synthesis algorithm for building self-adaptive software. *Information & Software Tech. 46,* 11, 719–727.

Valetto, G. and Kaiser, G. 2003. Using process technology to control and coordinate software adaptation. In *Proc. of Int. Conf. on Software Eng.* 262–273.

Verma, K. and Sheth, A. P. 2005. Autonomic web processes. In *Proc. of Int. Conf. on Service-Oriented Computing*. Lecture Notes in Computer Science, vol. 3826. 1–11.

Walsh, W. E., Tesauro, G., Kephart, J. O., and Das, R. 2004. Utility functions in autonomic systems. In *Proc. of IEEE Conference on Autonomic Computing*. 70–77.

WBEM. Web-based enterprise management standard. http://www.dmtf.org/standards/wbem/.

Westerinen, A., Schnizlein, J., Strassner, J., Scherling, M., Quinn, B., Perry, J., Herzog, S., Huynh, A.-N., and Carlson, M. 2000. Policy terminology. IETF, Internet Draft draftietf-policy-terminology-00.txt.

Weyns, D., Schelfthout, K., and Holvoet, T. 2005. Architectural design of a distributed application with autonomic quality requirements. *SIGSOFT Softw. Eng. Notes 30,* 4, 1–7.

White, J., Schmidt, D. C., and Gokhale, A. S. 2005. Simplifying autonomic enterprise java bean applications via model-driven development: A case study. In *Proc. of Int. Conf. on Model Driven Eng. Languages and Sys.* 601–615.

Willebeek-LeMair, M., Reeves, A., Center, I., and Heights, Y. 1993. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems 4,* 9, 979–993.

Woodside, C. M. and Menascé, D. A. 2006. Guest editors' introduction: Application-level QoS. *IEEE Internet Computing 10,* 3, 13–15.

Zhang, J. 2007. A formal approach to providing assurance to dynamically adaptive software. Ph.D. thesis, Michigan State University.