Modeling Dimensions of Self-Adaptive Software Systems

Jesper Andersson[1], Rogerio de Lemos[2], Sam Malek[3], Danny Weyns[4]

[1] Department of Computer Science, Växjö University,
S-351 95 Växjö Sweden
jesper.andersson@msi.vxu.se

[2] Computing Laboratory, University of Kent,
Canterbury, Kent, CT2 7NF, UK
r.delemos@kent.ac.uk

[3] Department of Computer Science, George Mason University,
MS 4A4, 4400 University Drive, Fairfax, VA, 22030 U.S.A.
smalek@gmu.edu

[4] Departement Computerwetenschappen, Katholieke Universiteit Leuven,
Celestijnenlaan 200 A, B-3001 Leuven, Belgium
danny.weyns@cs.kuleuven.be

Abstract. It is commonly agreed that a self-adaptive software system is one that can modify itself at run-time due to changes in the system, its requirements, or the environment in which it is deployed. A cursory review of the software engineering literature attests to the wide spectrum of software systems that are described as self-adaptive. The way self-adaptation is conceived depends on various aspects, such as the users' requirements, the particular properties of a system, and the characteristics of the environment. In this paper, we propose a classification of modeling dimensions for self-adaptive software systems. Each modeling dimension describes a particular facet of the system that is relevant to self-adaptation. The modeling dimensions provide the engineers with a common set of vocabulary for specifying the self-adaptive properties under consideration and select suitable solutions. We illustrate how the modeling dimensions apply to several application scenarios.

Keywords: Self-Adaptive, Self-*, Dynamic Adaptation, Modeling

1. Introduction

Over the past few decades we have witnessed an unrelenting pattern of growth in the size and complexity of software systems. This pattern of growth, which is very likely to continue well into the foreseeable future, has motivated software engineering researchers to develop techniques and tools that allow developers to deal with the complexity of designing, building and testing large-scale software systems. However, for the large part these advances have relied heavily on human reasoning or manual intervention.

At the same time, the emergence of highly distributed, mobile, and embedded systems that are often long-lived has made it increasingly infeasible to manually

manage and control such systems. This has prompted the development of a new class of software systems, namely self-adaptive software systems, which can modify their behavior at run-time due to changes in the system, its requirements, or the environment in which it is deployed. A cursory review of the software engineering literature attests to the wide spectrum of software systems that are argued to be self-adaptive. Indeed, there is a lack of consensus among researchers and practitioners on the points of variation among such software systems. We refer to these points of variations as *modeling dimensions*. The underlying insight guiding our study is that any self-adaptive system is built according to a conceptual model of adaptation, irrespective of the technologies and tools leveraged for its implementation. In fact, often the models of self-adaptation are represented implicitly in the form of domain knowledge or the engineer's expertise in the development of these systems. This in turn makes it harder to systematically, or even qualitatively, compare the different approaches.

In this paper, we identify modeling dimensions that describe various facets of self-adaptation, and classify these modeling dimensions in terms of four *groups*. This *classification* allows engineers to precisely specify the self-adaptive properties under consideration and select suitable solutions.

Note that it is not our ambition to be exhaustive, nor do we claim this is the only, or even the most appropriate classification. Our objective is to provide an initial impetus towards defining a comprehensive classification of key properties that are associated with self-adaptive systems. The purpose of such study is to establish a baseline from which key aspects of different self-adaptive system can be easily identified and compared. We demonstrate our classification's application in three different application domains. This exercise has served not only as a preliminary evaluation of the proposed classification, but has also helped us (the developers of these systems) to learn more about the specifics and in some cases intrinsically hidden characteristics of our systems. Finally, the classification of the modeling dimensions has aided us with identifying the current shortcomings of the state-of-the-art, which we propose to the software engineering community as future research challenges. We hope that it paves the way for focusing the future research efforts in this area.

The remainder of the paper is organized as follows. Section 2. describes an illustrative case, which serves as a motivating scenario for describing the classification of the modeling dimensions. Section 3. presents the details of the modeling dimensions. Section 4. discusses the application of the classification on two representative self-adaptive software systems. Section 5. provides an overview of open research challenges. Section 6. provides some pointers to related work. Finally, the paper concludes with a discussion of our contributions and our plans for extending this work.

2. Illustrative Case Study

As an illustrative case study, we consider the problem of obtaining dependable stock quotes from several, potentially unreliable, web sources [15]. The self-adaptation problem being considered is how to obtain reliable and available stock quotes through architectural reconfiguration. There are several web sources for stock quotes, for

example, Yahoo, Google, CNN, Reuters and FT, but these sources might not be available all the time, and there are no guarantees that the values that are being provided are correct, and moreover, their quality of services (QoS) may change. Based on the availability of resources, different fault-tolerant strategies, which rely on mechanisms, such as, voting, comparison and exception handling are employed in order to guarantee the delivery of dependable stock quotes. It is also assumed that the non-functional requirements (NFR) related to dependability may change during the system lifetime.

The system comprises (1) the application software that includes bridges for handling architectural mismatches and the fault tolerant strategies, (2) middleware that supports access to web services, and (3) the system infrastructure that includes computer hosts and the local area network. The user and the web sources for stock quotes are not considered to be part of the system.

3. Modeling Dimensions

We have grouped the identified key modeling dimensions for self-adaptive software systems into four groups: first, the dimensions associated with self-adaptability aspects of the system goals, second, the dimensions associated with causes of self-adaptation, third, the dimensions associated with the mechanisms to achieve self-adaptability, and fourth, the dimensions related to the effects of self-adaptability upon a system. Table 1 provides a summary of the modeling dimensions and their associated groups. Below we use different facets of the illustrative case study to exemplify the different modeling dimensions.

3.1 Goals

Goals are objectives the system under consideration should achieve [13]. Goals could either be associated with the lifetime of the system or with scenarios that are related to the system. Moreover, goals can either refer to the self-adaptability aspects of the application, or to the middleware or infrastructure that supports that application.

In the context of the case study mentioned above, amongst several possible goals, we consider, as an example, the following goal: "*the system shall deliver dependable (correct, responsive and available) stock quotes from the web*". This goal could be expressed in a way in which quantities are associated with the different attributes, and partitioned into sub-goals, with each sub-goal related to one of the attributes.

Evolution. This dimension captures whether the goals can change within the lifetime of the system. The number of goals may change, and the goals themselves may also change as the system as a whole evolves. Hence, goal evolution ranges from *static* in which changes are not expected, to *dynamic* in which goals can change at run-time, including the number of goals, i.e., the system is able to manage and create new goals during its lifetime.

In the context of the case study, the degree of goal evolution is static because a goal is not expected to change at run-time. However, if some stock quote providers start to

charge for their services, then a new goal could be introduced to accommodate the need of the system to look for free services.

Flexibility. This dimension captures whether the goals are flexible in the way they are expressed [4]. This dimension is related to the level of uncertainty associated with the goal specification, which may range over three values: *rigid*, *constrained*, and *unconstrained*. A goal is rigid when it is prescriptive, while a goal is unconstrained when its statement provides flexibility for dealing with uncertainty. An example of a rigid goal is "*the system shall do this…*", while an unconstrained goal is "*the system might do this…*" A constrained goal provides a middle ground, where there is flexibility as long as certain constraints are satisfied, such as, "*the system may do this… as long as it does this…*"

In the context of the case study, the goal as stated is rigid. However, if we consider a scenario in which the non-functional requirements (NFR) associated with a goal can change according to the quality of services (QoS) of the resources available, then the goal in terms of flexibility could be considered unconstrained. For example, if the NFR associated with the goal cannot be achieved, then the goal can be relaxed through some best effort analysis.

Duration. This dimension is concerned with the validity of a goal throughout the system's lifetime. It may range from *temporary* to *persistent*. While a persistent goal should be valid throughout the system's lifetime, a temporary goal may be valid for a period of time: *short*, *medium* and *long term*. A persistent goal may restrict the adaptability of the system because it may constrain the system flexibility in adapting to change. A goal that is associated with a particular scenario can be considered a temporary goal.

In terms of duration, the goal of the illustrative case can be considered persistent since it is related with the purpose of the system. On the other hand, a temporary goal could be "*the system shall deliver stock quotes more often when the volume of transactions go above a certain threshold*".

Multiplicity. This dimension is related to the number of goals associated with the self-adaptability aspects of a system. A system can either have a *single* goal or *multiple* goals. As a general rule of thumb, a single goal self-adaptive system is relatively easier to realize than systems with multiple goals. As discussed in the next dimension, this is particularly true for system where the goals are related.

In the illustrative case, since there are several NFRs associated with the system's overall objective, there are several goals that need to be satisfied. Therefore, we characterize the multiplicity dimension as multiple.

Dependency. In case a system has multiple goals, this dimension captures how the goals are related to each other. They can be either *independent* or *dependent*. A system can have several independent goals (i.e., they don't affect each other). When the goals are dependent, goals can either be *complementary* with respect to the objectives that should be achieved or they can be *conflicting*. In the latter case, tradeoffs have to be analyzed for identifying an optimal configuration of the goals to be met.

In the illustrative example, the goals that are extracted from the main objective can be considered dependent. Moreover, if cost is introduced as a NFR, then the goals can be considered as conflicting since those web sources that are able to provide better QoS might have a higher associated cost.

Table 1. Modeling dimensions for self-adaptive software systems.

Dimensions	*Degree*	*Definition*
Goals – goals are objectives the system under consideration should achieve		
evolution	static to dynamic	whether the goals can change within the lifetime of the system
flexibility	rigid, constrained, unconstrained	whether the goals are flexible in the way they are expressed
duration	temporary to persistent	validity of a goal through the system lifetime
multiplicity	single to multiple	how many goals there are?
dependency	independent to dependent (complementary to conflicting)	how the goals are related to each other
Change – change is the cause for adaptation		
source	external (environmental), internal (application, middleware, infrastructure)	where is the source of change?
type	functional, non-functional, technological	what is the nature of change?
frequency	rare to frequent	how often a particular change occurs?
anticipation	foreseen, foreseeable, unforeseen	whether change can be predicted
Mechanisms – what is the reaction of the system towards change		
type	parametric to structural	whether adaptation is related to the parameters of the system components or to the structure of the system
autonomy	autonomous to assisted (system or human)	what is the degree of outside intervention during adaptation
organization	centralized to decentralized	whether the adaptation is done by a single component or distributed amongst several components
scope	local to global	whether adaptation is localized or involves the entire system
duration	short, medium, long term	how long the adaptation lasts
timeliness	best effort to guaranteed	whether the time period for performing self-adaptation can be guaranteed
triggering	event-trigger to time-trigger	whether the change that triggers adaptation is associated with an event or a time slot
Effects – what is the impact of adaptation upon the system		
criticality	harmless, mission-critical, safety-critical	impact upon the system in case the self-adaptation fails
predictability	non-deterministic to deterministic	whether the consequences of adaptation can be predictable
overhead	insignificant to failure	the impact of system adaptation upon the quality of services of the system
resilience	resilient to vulnerable	the persistence of service delivery that can justifiably be trusted, when facing changes

3.2 Change

Changes are the cause of adaptation. When there is a change in the system, its requirements, or the environment in which it is deployed, this may cause the system to self-adapt. There are changes in which the system is expected to act upon, while others can be masked from the system. Changes can be classified in terms of place in which change has occurred, the type and the frequency of the change, and whether it can be anticipated. All these elements are important for identifying how the system should react to change that occurs during run-time.

In the context of the illustrative case study mentioned above, we consider the cause of adaptation to be the failure of web sources, the reduced QoS from web sources, and changes in the NFR (expressed as goals) associated with the system.

Source. This dimension identifies the origin of the change, which can be either *external* to the system (i.e., its environment) or *internal* to the system, depending on the scope of the system. In case the source of change is internal, it might be important to identify more precisely where change has occurred: *application*, *middleware* or *infrastructure*.

The source of the two changes related to the service providers is external to the system. The change of Apache version, on which the application runs, is an internal change that happens in the middleware.

Type. This dimension refers to the nature of change. It can be *functional*, *non-functional*, and *technological*. Technological refers to both software and hardware aspects that support the delivery of the services. Examples of the three types of change are, respectively: the purpose of the system has changed and services delivered need to reflect this change, system performance and reliability need to be improved, and the version of the middleware in which the application runs has been upgraded.

In the illustrative case, since the changes are related to the QoS of the web sources, the type of change is non-functional, and the failure of a web source is also considered a non-functional change. An example of a technological change is the upgrade of the Apache version.

Frequency. This dimension is concerned with how often a particular change occurs, and it can range from *rare* to *frequent*. If for example a change happens quite often this might affect the responsiveness of the adaptation.

Failures in the web sources are expected to occur quite often, hence the frequency of change is frequent. On the other hand, if we consider changes in NFR, these should be quite rare to occur.

Anticipation. This dimension captures whether change can be predicted ahead of time. Different self-adaptive techniques are necessary depending on the degree of anticipation: *foreseen* (taken care of), *foreseeable* (planned for), and *unforeseen* (not planned for) [14].

Although faults are undesirable, they should be expected to occur, hence the failure of a web resource should be considered as foreseen. In contrast, the upgrade of the Apache should be considered as a foreseeable change, and the provision of dependable weather forecast instead of stock quotes should be considered as unforeseen.

3.3 Mechanisms

This set of dimensions captures the system reaction towards change, which means that they are related to the adaptation process itself. The dimensions associated with this group refer to the type of self-adaptation that is expected, the level of autonomy of the self-adaptation, how self-adaptation is controlled, the impact of self-adaptation in terms of space and time, how responsive is self-adaptation, and how self-adaptation reacts to change.

In the context of the illustrative case study mentioned earlier, we consider the mechanism for self-adaptation to be the system's architectural reconfiguration in which the structure of the system is modified as a means to accommodate change.

Type. This dimension captures whether adaptation is related to the parameters of the system's components or to the structure of the system. Based on this, adaptation can be *parametric* or *structural*, or a combination of these. Structural adaptation could also be seen as compositional, since it depends on how components are integrated (e.g., dynamic weaving [20]).

The type of self-adaption considered in the illustrative case study is structural since configurations are changed and components and connectors are replaced. An example of structural adaptation is when a configuration based on majority voting has to be changed to a configuration based on comparison because of the lack of resources.

A parametric type self-adaptation would be to increase the time interval between two stock quote readings.

Autonomy. This dimension identifies the degree of outside intervention during adaptation. The range of this dimension goes from *autonomous* to *assisted*. In the autonomous case, at run-time there is no influence external to the system guiding how the system should adapt. On the other hand, a system can have a degree of self-adaptability when externally assisted, either by another system or by human participation (which can be considered another system).

In the illustrative case, for the foreseen type of changes the system is autonomous, but for the foreseeable type of changes, such as a change in the Apache version, human participation is likely to be required.

Organization. This dimension captures whether adaptation is performed by a single component – *centralized*, or distributed amongst several components – *decentralized*. If adaptation is decentralized no single component has a complete control over the system.

The self-adaptation in the case study relies on a complete model of the system, hence the organization is centralized.

Scope. This dimension identifies whether adaptation is localized or involves the entire system. The scope of adaptation can range from *local* to *global*. If adaptation affects the entire system then more thorough analysis is required to commit the adaptation. It is fundamental for the system to be well structured in order to reduce the impact that change might have on the adaptation.

In the illustrative case, the current architectural configuration of the system and the web resource that has failed determines the scope of the self-adaptation. For instance, it may be global if it involves the reconfiguration of the whole system to come up with a new fault tolerance strategy.

Duration. This dimension refers to the period of time in which the system is self-adapting, or in other words, how long the adaptation lasts. The adaptation process can be for *short* (seconds to hours), *medium* (hours to months), or *long* (months to years) term. Note that time characteristics should be considered relative to the application domain. While scope dimension deals with the impact of adaptation in terms of space, duration deals with time.

Considering that the time it takes for architectural reconfiguration is minimal (in the scale of seconds) when compared with the lifetime of the system (months), the duration of the self-adaptation in the context of the case study should be short term.

Timeliness. This dimension captures whether the time period for performing self-adaptation can be guaranteed, and it ranges from *best-effort* to *guaranteed*. For example, in case change occurs quite often, it may be the case that it is impossible to guarantee that adaptation will take place before another change occurs, in these situations best effort should be pursued.

In the context of the case study, upper bounds on the process of architectural reconfiguration can be easily identified, hence the timeliness associated with self-adaptation can be guaranteed.

Triggering. This dimension identifies whether the change that initiates adaptation is *event-trigger* or *time-trigger*. Although it is difficult to control how and when change occurs, it is possible to control how and when the adaptation should react to a certain change. If the time period for performing adaptation has to be guaranteed, then an event-trigger might not provide the necessary assurances when change is unbounded.

In the illustrative case, the self-adaption mechanism is event-triggered, when a fault occurs, it is detected and the system starts the process of architectural reconfiguration.

3.4 Effects

This set of dimensions capture what is the impact of adaptation upon the system, that is, it deals with the effects of adaptation. While mechanisms for adaptation are properties associated with the adaptation, these dimensions are properties associated with system in which the adaptation takes place. The dimensions associated with this group refer to the criticality of the adaptation, how predictable it is, what are the overheads associated with it, and whether the system is resilient in the face of change.

In the context of the illustrative case study mentioned earlier, we consider that the system fails if it is not able to provide dependable stock quotes.

Criticality. This dimension captures the impact upon the system in case the self-adaptation fails. There are adaptations that harmless in the context of the services provided by the system, while there are adaptations that might involve the loss of life. The range of values associated with this criticality is *harmless*, *mission-critical*, and *safety-critical*.

The level of criticality of the application (and the adaptation process) is mission-critical, since it may lead to some financial losses.

Predictability. This dimension identifies whether the consequences of self-adaptation can be predictable both in value and time. While timeliness is related to the adaptation mechanisms, predictability is associated with system. Since predictability is

associated with guarantees, the degree of predictability can range from *non-deterministic* to *deterministic*.

Given that in the illustrative case there are no guarantees sufficient web sources will be available for the continued provisioning of services, the predictability of the adaptation is non-deterministic.

Overhead. This dimension captures the negative impact of system adaptation upon the system's performance. The overhead can range from *insignificant* to system *failure* (e.g., thrashing). The latter will happen when the system ceases to be able to deliver its services due to the high-overhead of running the self-adaptation processes (monitoring, analyzer, planning, effecting processes).

Since the architecture of the system that provides dependable stock quotes is based on web services, the overall overhead associated with the architectural reconfiguration is quite reasonable. In other words, although the system ceases to provide services for some time interval, this interval is acceptable.

Resilience. This dimension is related to the persistence of service delivery that can justifiably be trusted, when facing changes [14]. There are two issues that need to be considered under this dimension: first, it is the ability of the system to provide resilience, and second, it is the ability to justify the provided resilience. The degree of resilience can range from *resilient* to *vulnerable*.

In the context of the illustrative case study, the system is resilient to certain types of change (failures of web sources) because the self-adaptation which is responsible for the continuous provisioning of services can be analyzed for extracting the assurances that are needed for justifying resilience.

4. Evaluation – Case studies

A classification framework is generally difficult to evaluate, mainly due to the process used to develop the classification. A formal evaluation, such as the one proposed by Gómez-Pérez [8], would require a formal specification of our classification framework, which is not feasible for our topic of study. Therefore, we adopt a more practical approach to validate our classification framework.

The main contribution of our work is the descriptive model of the modeling dimensions for self-adaptive software systems. The evaluation of the proposed classification was conducted through applying it to several previously developed self-adaptive software systems. The case studies represent different classes of application domains: (1) Traffic Jam Monitoring Systems [9], (2) Embedded Mobile Systems [16,17,19], and (3) High Performance Computing and Sensor Networks [1]. The feedback from the case studies improved the classification in several iterations. Due to space constraints we only present the results of the first two studies below.

We use the classification in two different ways, In the Traffic Monitoring System, we apply the various modeling dimensions to a single scenario of self-healing. This approach provides detailed insight on one particular quality property of the system. In the Embedded mobile System, the modeling dimensions are applied to multiple QoS concerns. This approach provides insight on a set of related quality properties of the system.

4.1 Traffic Jam Monitoring System

Intelligent transportation systems (ITS) refer to systems that utilize advanced information and communication technologies to improve the safety, security and efficiency of transportation systems [6,9]. One particularly challenging problem in traffic is congestion. A first step to address this problem is monitoring the traffic. We describe an agent-based approach for traffic monitoring that enables the detection of traffic jams in a decentralized way, avoiding the bottleneck of a centralized control center. Our interest is in a particular scenario of self-healing that allows the system to deal with silent node failures (i.e., a type of failure that occurs when the failing node becomes unresponsive without sending any incorrect data). We introduce the application and briefly explain how self-healing is added to the system. Then we give an overview of the modeling dimensions for the self-healing scenario.

4.1.1 Application

The traffic monitoring system consists of a set of intelligent cameras which are distributed evenly along a highway. Each camera has a limited viewing range and cameras are placed to get an optimal coverage of the highway. A camera is able to measure the current congestion level of the traffic and decide whether there is a traffic jam or not in its viewing range. Each camera is equipped with a communication unit to interact with other cameras. The task of the cameras is to detect and monitor traffic jams on the highway in a decentralized way, i.e. without any centralized entity involved. Possible clients of the monitoring system are traffic light controllers and driver assistance systems such as systems that inform drivers about expected travel time delays. Since traffic jams can span the viewing range of multiple cameras and can dynamically grow and dissolve, the data observed by multiple cameras has to be aggregated. Without a central point of control, cameras have to collaborate and distribute the aggregated data to the clients. To support such dynamic organizations, we have applied an agent-based design for the system [9]. On each camera an agent is deployed that can play different roles in organizations. Example roles are “data pusher” and “data aggregator.” Agents exploit a distributed middleware which provides support for dynamic organizations. The middleware encapsulates the management of dynamic evolution of organizations offering possible roles to agents based on the current context. Figure 1 shows the deployment view of the agent-based traffic monitoring system.

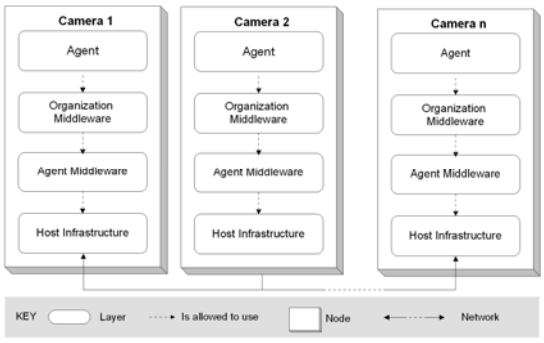

Figure 1. Deployment view of the traffic monitoring system.

The software on each camera is structured in layers. The Host Infrastructure layer encapsulates common middleware services and basic support for distribution, hiding the complexity of the underlying hardware. The Agent Middleware layer provides basic services in multi-agent systems [22], including support for perception, action, and communication. The Organization Middleware layer provides support for dynamic organizations. The layer encapsulates the management of dynamic evolution of organizations and it provides role-specific services to the agents for perception, action, and communication. Finally the Agent layer encapsulates the agents that provide the associated functionality in the organizations for monitoring traffic jams.

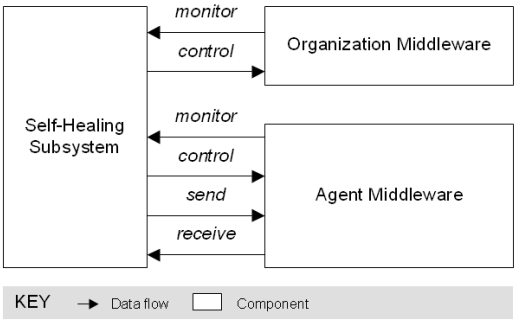

Figure 2. Integration of the self-healing subsystem with the organization middleware and agent middleware on one node

4.1.2 Self-Healing

When a node fails, the system may enter an inconsistent state in which agents and the organization middleware are no longer capable of working according to their specification. To deal with this kind of failures, an additional self-healing subsystem (SHS) is deployed on each node. The SHS interacts with the local agent middleware and organization middleware, and relies on the functionalities provided by the agent middleware to interact with SHSs on other nodes. Figure 2 shows the integration of the self-healing subsystem with the system software on one node.

SHSs periodically exchange alive signals using the communication service of the agent middleware (send and receive). Node failures are detected by monitoring the alive signals. When a SHS detects a failure, it adapts the local structure of the organizations in which the agent of the failing node is involved in and possibly interacts with SHSs on other involved nodes to bring the system in a consistent state from which it can continue its function in a degraded mode.

4.1.3 Modeling Dimensions

We consider the scenario where the system is in normal operation mode (camera agents are collecting data and provide information to clients about possible traffic jams) and one of the nodes fails silently. Such event may result in corrupt organizations with lost or missing roles on the failed node. The self-healing subsystem

(SHS) detects the failure and restores the system to a consistent state so that it can continue its operation.

Goals – The system shall recover from a silent node failure and continue its operation in a degraded mode.

- **Evolution**: *static* – Recovering from silent node failures is a goal that will not change over the life time of the system.
- **Flexibility**: *rigid* – A node failure compromises the consistency of the system and as such it threatens service delivery. In order to remain operational, the system must deal with node failures.
- **Duration:** *persistent* – Silent node failures can occur at any time during normal operation. As such, recovering from silent node failures is a persistent goal.
- **Multiplicity:** *multiple* – Besides dealing with node failures, the system has other goals as well. The primary goal of the system is to deliver a monitoring service to clients interested in traffic jams. Other goals refer to particular qualities of the system such as accuracy of observation and reaction time.
- **Dependency:** *dependent* – There is a dependency between the self-healing goal and the delivery of services. If the system fails to recover from a node failure, the quality of the services will significantly degrade.

Change – a node fails silently.

- **Source:** *external* (environment) and *internal* (application) – A silent node failure can be caused by an external trigger such as a hardware failure, or it can be caused by a crash of the software running on the node.
- **Type:** *technological* – The cause for self-adaptation is of a technologic nature: a node in the system fails. If the system reacts not properly, the failure will harm the system functionality.
- **Frequency:** *rare* – Silent node failures happen rarely.
- **Anticipation:** *foreseen* – Neither the place nor the time of a silent node failure can be predicted. Still, the system can anticipate how to react when a silent node failure occurs.

Mechanisms – the SHSs restore the system in a consistent state.

- **Type:** *parametric / structural* – From the point of view of a single node which is involved in a failure, the adaptation is parametric since the SHS will restore the affected local state of the system. From the point of view of the system, adaptation is structural since the changes applied by the SHSs on the nodes involved in a failure will change the structure of collaborating cameras (i.e. the failing camera will no longer be part of the collaboration).
- **Autonomy:** *autonomous* – The SHS acts fully autonomously. The self-adaptation process will take place without a human involved. However, restoring the failed node typically will require human intervention.
- **Organization:** *decentralized* – SHSs deployed on the different nodes collaborate to detect a node failure. The required adaptations are performed locally. No central monitor or controller is involved.
- **Scope:** *local* – The adaptation is performed locally. Only the nodes with cameras taking part in organizations with the camera of the failed node will be involved in the adaptation process.

- **Duration:** *short term* – The adaptation process should be completed in seconds. This is orders of magnitude faster as traffic jams arise or dissolve.
- **Timeliness:** *best effort* – The time period required for performing the adaptation depends on several factors, such as the current traffic conditions and the available bandwidth. Given the relative short duration of the adaptation (comparing to the duration of the traffic jam phenomena), best effort meets the required timeliness.
- **Triggering:** *event-trigger* – Adaptation is triggered by the detection of missing alive messages exchanged between SHSs.

Effects – the system will continue its functionality in degraded mode.

- **Criticality:** *harmless* – The services provided by the traffic monitoring system are in general not critical. If the adaptation fails, the functionality of the system may significantly degrade, however, no human lives are involved.
- **Predictability:** *deterministic* – The consequences of a node failure are clear. The information provided by the failed node will no longer be available. The SHSs will bring the system in a consistent state so that it can continue its operation.
- **Overhead:** *almost insignificant* – After adaptation, the quality of the services provided by the system will slightly degrade in case a traffic jam occurs in the neighborhood of the failed node. All traffic information collected outside the range of the camera of the failed node will not be affected.
- **Resilience:** *semi-resilient* – After adaptation, service delivery will persist with only minimal decrease of quality. In case of repeatable node failures, the quality of service delivery may become seriously affected, in particular when neighboring nodes fail.

4.2 Embedded Mobile System

Below we present the application of our classification model to another self-adaptive software system, which is representative of an emerging class of mobile, pervasive, and cyber physical systems. These systems are inherently different from traditional software systems. For instance, network failures and changes in the availability of resources are considered the norm, instead of an exception. As detailed further below, self-adaptation has been shown as a promising approach to deal with the unpredictability of such systems.

4.2.1 Application

Emergency Deployment System (EDS) is a mobile application intended for distributed management and deployment of personnel to deal with situations such as natural disasters and search-and-rescue efforts. An instance of EDS (shown in Figure 3) consists of *Headquarters*, *Team Leader*, and *Responder* applications that leverage the software services and wireless sensors provided by the system to achieve their tasks. The *Headquarters* computer is networked via secure links to a set of mobile devices used by *Leaders* during the operation. Each *Leader* is capable of controlling his own part of the crisis scene: deploying *Responders*, analyzing the deployment strategy, transferring *Responders* between *Leaders*, and so on. *Responders* can only

view the segment of the operation in which they are located, receive direct orders from the *Leaders*, and report their status.

The domain of emergency and response is by its nature unpredictable. For instance, it is completely conceivable that due to some unforeseen event the *Headquarters* fail, in which case it is desirable for a designated *Leader* to assume the role of the *Headquarters*. On top of the unpredictability of the application domain, given that EDS is a mobile platform, it also needs to be able to deal with fluctuations in the availability of computing resources. For instance, the system should be able to deal with situations where as a result of user mobility the network connectivity or its throughput changes significantly.

Figure 3. An instance of EDS application.

4.2.2 Self-Adaptation

In response to the Quality of Service (QoS) challenges of mobile software systems, such as those faced by EDS, software engineers have previously developed a variety of run-time adaptation techniques, including caching [11] or hoarding [12] of data, and multi-modal components [21]. In our work [16,17,19] we have demonstrated that a software system's *deployment architecture* (i.e., allocation of the system's software components to its hardware hosts) has a significant impact on the mobile system's QoS properties. For example, a mobile system's latency can be improved if the system is deployed such that the most frequent and voluminous interactions among the components involved in delivering the functionality occur either locally or over reliable and capacious wireless network links. A key observation is that most system parameters (e.g., available bandwidth, reliability of networks, and frequency of interactions) that are required for finding a good deployment architecture are not known until run-time, and even then they can change. Therefore, a redeployment of the software system via migration of its components may be necessary to improve its QoS.

We have developed a self-adaptive infrastructure [16,17,19] for improving a mobile system's deployment architecture that consists of four phases: 1) monitoring the system parameters of interest (e.g., reliability of links, frequencies of interaction), 2) populating a deployment model of the system with the monitored system properties, 3) finding a new deployment architecture that improves the system's QoS properties, and 4) effecting the new deployment architecture via software component mobility [7]. In order to reason about multiple QoS dimensions we leverage a multivariate utility function. The utility function allows us to resolve the trade-offs among multiple QoS

dimensions (e.g., when improvement in one QoS results in degradation in another QoS).

4.2.3 Modeling Dimensions

We have applied the above approach for improving a mobile software system's deployment architecture on several instances of EDS. Below we provide an analysis of this work in the context of the modeling dimensions from Section 3. :

Goals – improve the users' preference, which is specified in the form of a utility function.

- **Evolution:** *dynamic* – New QoS concerns may be added, old ones may be removed or modified, and the users' preferences for the QoS concerns may change.
- **Flexibility:** *constrained* – The goal is to maximize the utility function as long as the system constraints are satisfied. An example of a system constraint is as follows: the amount of memory required for software components that are deployed on a host should be less than the amount of available memory on that host.
- **Duration:** *long term* – The system is always in pursuit of the optimal configuration.
- **Multiplicity:** *multiple* – Since most instances of EDS consist of multiple users with different roles (e.g., commanders, leaders, troops), and the fact that often there are multiple QoS dimensions of importance to each user, the goals are often multi-faceted.
- **Dependency:** *dependent* – The goals are dependent on one another. For example, a deployment architecture that maximizes the system's security often leverages complex encryption and authentication protocols, which have a negative impact on the system's latency.

Change – fluctuations in system parameters (e.g., network bandwidth, reliability) and changes in the system usage (i.e., load).

- **Source:** *external* (environment) and *internal* (application) – Source of change could be either external environment, such as a situation when a mobile user's connectivity is impacted severely due to his movement (e.g., when the user is behind thick walls). Alternatively, the source of change could be internal application, where some of the distributed components interact more frequently than others. In this case, a better deployment may be to collocate the components to minimize the amount of remote communication.
- **Type:** *non-functional* – Changes in the system could potentially degrade the level of QoS provisioned by the system.
- **Frequency:** *frequent* – System parameters are changing constantly. However, in order to avoid the system from constantly redeploying itself, changes in the system are observed over a period of time, and only changes that are significant enough are reported to the adaptation modules.
- **Anticipation:** *foreseen* – In mobile systems changes in system parameters are the norm, rather than the exception. In EDS we foreseen such changes, and developed the appropriate mitigation capability.

Mechanisms – redeployment of software components.

- **Type:** *structural* – Through component redeployment the deployment architecture of the system is changed.
- **Autonomy:** *autonomous* – EDS is a long-lived and highly distributed system. At the same time, given that changes in such a system are frequent and unpredictable, it is infeasible for a manual control of adaptation at run-time.
- **Organization:** *centralized* analysis, *decentralized* adaptation – Finding (calculating) a new optimal deployment architecture is performed centrally, effecting the actual change (i.e., migrating and rebinding software components) is performed by individual platforms in a decentralized manner.
- **Scope:** *local* and *global* – The actual components that are redeployed depend on the results of the analysis. The result of adaptation could range from redeployment of a single software component to redeployment of the entire system.
- **Duration:** *short term* – The amount of time required to redeploy the system should be short. Redeployment impacts the availability of (some of) the system's services. This is in particular true for the EDS system that has stringent availability requirement.
- **Timeliness:** *best effort* – The time required for adaptation depends on a number of system parameters (e.g., network throughput) as well as the sizes of the software components that need to be redeployed. Therefore, it is not possible to provide any hard guarantees. For relatively stable systems it is feasible to determine a time bound.
- **Triggering:** *event-trigger* – The triggering usually depends on the patterns identified in the monitored data. If the monitored data indicates significant changes in the system, the analysis process is initiated.

Effects – system provisions its services with higher level of QoS.

- **Criticality:** *mission-critical*, *safety-critical* – Depending on the nature of emergency scenario EDS could be either a mission or safety critical system.
- **Predictability:** *non-deterministic* – An underlying assumption in EDS is that changes in the past are a good indicator of the system's future behavior. However, this assumption may not hold, in particular if the users' usages of the system's services or its parameters change dramatically.
- **Overhead:** *reasonable* – In EDS we have developed a mechanism to ensure the system does not constantly redeploy itself. This is realized by ensuring that the adaptation is triggered only if there are significant changes in the monitored data over a prespecified period of time. However, there is a considerable overhead in terms of wasted resources (e.g., battery) for the redeployment of the components, and if the components are large this overhead may be prohibitive.
- **Resilience:** *semi-resilient* – A services (functionality) becomes temporary unavailable if some of the software components involved in provisioning that service are currently being redeployed. However, there is no impact to the services that do not depend on the part of the system that is being redeployed.

5. Challenges of Modeling Self-Adaptive Systems

Substantial progress has been made by the software engineering community in tackling the challenges posed by each of the discussed modeling dimensions. However,

there are several important research questions that are remaining. Our study of the modeling dimensions, in particular the exercise of applying it to several self-adaptive software systems, helped us to identify some important research questions that should be the focus of future research in this area. We briefly elaborate on those below based on the categories of the modeling dimensions.

5.1 Goals

A self-adaptive software system often needs to perform a trade-off analysis between several potentially conflicting goals. Current state-of-the-art techniques leverage a utility function to map the trade-offs among several conflicting goals to a scalar value, which is then used for making decisions about adaptation. However, in practice, defining such a utility function is a challenging task. Practical techniques for specifying and generating utility functions, potentially based on the user's requirements, are needed. One promising direction is to use preferences that compare situations under Pareto optimal conditions.

5.2 Change

Often the adaptation is triggered by the occurrence of a pattern in the data that is gathered from a running system. For example, the system is monitored to determine when a particular level of QoS is not satisfied, which then initiates the adaptation process. However, monitoring a system, especially when there are several different QoS properties of interest, has an overhead. In fact, the amount of degradation in QoS due to monitoring could outweigh the benefits of improvements in QoS to adaptation. We believe that more research on light-weight monitoring techniques, as well as more advanced models that take the monitoring overhead of the approach into account are needed.

5.3 Mechanisms

Many types of adaptation techniques have been developed: architecture-based adaptation that is mainly concerned with structural changes at the level of software components, parametric based adaptation that leverages policy files or input parameters to configure the software components, aspect-oriented-based adaptation that changes the behavior of a running system via dynamic weaving techniques. Researchers and practitioners have typically leveraged a single tactic to realize adaptation based on the characteristics of the target application. However, given the unique benefits of each approach, we believe a fruitful avenue of future research is a more comprehensive approach that leverages several adaptation tactics simultaneously.

Most state-of-the-art adaptive systems are built according to the centralized control loop pattern. Thereby, if applied to a large-scale software system, many such techniques suffer from scalability problems. The field of multi-agent systems has developed a large body of knowledge on decentralized systems, where each agent (software component) adapts its behavior at run-time. Related are biologically inspired adaptation systems that tend to further push the limits of decentralization. While these

approaches are promising, practical experiences with these approaches in real-world settings are limited. Methods used in systems engineering, like hierarchical organization and coordination schemes could also be applicable. There is a pressing need for decentralized, but still manageable, efficient, and predictable techniques for constructing self-adaptive software systems. A major challenge is to accommodate a systematic engineering approach that integrates both control-loop approaches with decentralized agent inspired approaches.

Responsiveness is a crucial property in real-time software systems, which are becoming more prevalent with the emergence of embedded and cyber-physical systems. These systems are often required to deterministically respond within pre-specified (often short) time intervals, making it extremely difficult to adapt the system, while satisfying the deadline requirements. There is a need for adaptation models targeted for real-time systems that treat the duration and overhead of adaptation as first class entities.

5.4 Effects

Adapting safety-critical software systems, while ensuring the safety requirements, has remained largely an out-of-reach goal for the practitioners and researchers. There is a need for verification and validation techniques that guarantee safe and sound adaptation of safety-critical systems, under all foreseen and unforeseen events.

Finally, predicting the exact behavior of a software system due to run-time changes is a challenging task. For example, while it may be possible to predict the new functionality that will become available as a result of replacing a software component, it may not be possible to determine what will be the impact of the replaced software component on the other components that are sharing the same resources (e.g., CPU, memory, and network). More advanced and predictive models of adaptation are needed for systems that could fail to satisfy their requirements due to side-effects of change.

In highly dynamic systems, such as mobile systems, where the environmental parameters change frequently, the overhead of adaptation due to frequent changes in the system could be so high that the system ends up thrashing. This overhead includes the frequent execution of the reasoning algorithms (e.g., finding a new configuration), the downtime of a portion of the system due to making changes, or simply the resource cost (e.g., wasted CPU cycles, battery power) of changing the system. The trade-offs between the adaptation overhead and the accrued benefits of changing the system needs to be taken into consideration for such systems.

6. Related Work

This work defines a classification of modeling dimensions that should be considered when modeling self-adaptive software systems. Similar classifications exist but our survey reveals that none is suitable for characterizing the modeling variations among self-adaptive software systems. Dobson et al. provide a survey of techniques applied to autonomic computing [5]. Buckley et al. define a taxonomy for software change [3], which unlike our approach is not focused on run-time adaptation (change) of software. Another major difference is that goals are not made explicit in Buckeley's taxonomy.

Implementation of adaptability requires support from middleware or languages, Bradbury et al. classify support from dynamic software architecture languages [2]. This work has a focus on architecture specification and does not consider the goals. Similarly, the taxonomy by McKinley et al. [18] specifically targets compositional software adaptation, and is not applicable to other types of self-adaptive software. The work by Laprie on a classification of resilience [14] has also inspired some of the dimensions and their values in our work.

7. Discussions and Future Work

The classification model presented in this paper applies to any self-adaptive software system. We believe that this classification will be useful in several different development situations. It can be used as a driver for traditional forward engineering, but also useful in a reverse engineering context where engineers comprehend the existing solutions.

The classification brings more structure to the area of self-adaptive software systems. With the classification in mind it is more likely that an individual engineer as well as groups of engineers to be able to understand the technology domain better, thus avoiding situations where two or more interpretations of a technique affect the development process. Our intention with the classification has been to create a vocabulary that can be used within, for instance, a design team.

The classification can also be used to drive development. Despite its rather high level of abstraction, the groups, dimensions, and degrees can be used as a requirements statement for the self-adaptive scenarios in a system. This information supports decision making about tools, languages, and middleware platforms.

The classification could also be utilized in reverse engineering activities. Part of this process is “understanding structures” that are currently present in a self-adaptive application, and then go forward and change. The classification provides a check-list for conceptual and physical concepts concerned with structural and behavioral properties that can be identified in existing application documentation, hence assist in classifying an application’s self-adaptive behavior.

While our experience with the classification model has been positive so far, we believe the classification model can be refined further. In particular, we would like to provide a more detailed enumeration of possible values for the classification’s degree attribute (i.e., the middle column of Table 1). We also hypothesize that the majority of self-adaptive software systems are developed according to a handful of architectural patterns. We intend to leverage the proposed classification model, which allows for systematically identifying the variations among different self-adaptive software systems, to study and document the most commonly used architectural patterns for such systems.

8. References

1. J. Andersson, et al. An Adaptive High-Performance Service Architecture. *ETAPS Workshop on Software Composition Electronic Notes Theoretical Computer Science* 114. 2005.
2. J. S. Bradbury, et al. A Survey Of Self-Management In Dynamic Software Architecture Specifications. *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems (WOSS '04)*. Eds. D. Garlan, J. Kramer, and A. Wolf. pp. 28-33.

3. J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards A Taxonomy of Software Change. *Journal of Software Maintenance and Evolution*. Sep 2005. pp. 309-332.
4. B. H. C. Cheng, et al. Software Engineering for Self-Adaptive Systems: A Research Road Map. *Software Engineering for Self-Adaptive Systems. Proceedings of the 08031 Dagstuhl Seminar*. Eds. B. H. C. Cheng et al. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. 2008.
5. S. Dobson, S. Denazis, et al. A Survey of Autonomic Communications. *ACM Transactions on Autonomous and Adaptive Systems*, vol 1, no 2, pp 223-259, 2006.
6. ERTICO. Intelligent Transportation Systems for Europe, http://www.ertico.com/.
7. A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Software Engineering* 24. 1998. pp. 342-361.
8. A. Gómez-Pérez. Evaluation of Ontologies. *International Journal of Intelligent Systems 16.* 2001. pp. 391-409.
9. R. Haesevoets, et al. Managing Agent Interactions With Context-driven Dynamic Organizations. *Engineering Environment-Mediated Multi-Agent Systems*. Lecture Notes in Computer Science, vol. 5049, 2007.
10. ITS. Intelligent Transportation Society of America, http://www.itsa.org/.
11. J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems* 10(1). Feb 1992.
12. G. H. Kuenning and G. J. Popek. Automated Hoarding for Mobile Computers. *ACM Symp. on Operating Systems Principles*. St. Malo, France. Oct 1997.
13. A. van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. *Proceedings of 5th IEEE International Symposium on Requirements Engineering (RE'01)*. Toronto, Canada. Aug 2001. pp. 249-263.
14. J. C. Laprie. From Dependability to Resilience. *Supplemental Proceedings of the International Conference on Dependable Systems & Networks (DSN 2008).* Anchorage, Alaska. June 2008. pp. G8-G9.
15. R. de Lemos. Architecting Web Services Applications for Improving Availability. *Architecting Dependable Systems III*. Eds. R. de Lemos, C. Gacek, A. Romanovsky. LNCS 3549. Springer. Berlin, Germany. 2005. pp. 69-91.
16. S. Malek, et al. A Framework for Ensuring and Improving Dependability in Highly Distributed Systems. *Architecting Dependable Systems III*. Eds. R. de Lemos, C. Gacek, A. Romanovsky. LNCS 3549. Springer. Berlin, Germany. 2005.
17. S. Malek, C. Seo, S. Ravula, B. Petrus, and N. Medvidovic. Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support. *International Conference on Software Engineering (ICSE 2007)*. Minneapolis, Minnesota, May 2007.
18. P. K. Mckinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing Adaptive Software. *IEEE Computer* 37(7). July 2004. pp. 56-64.
19. M. Mikic-Rakic, S. Malek, and N. Medvidovic. Architecture-Driven Software Mobility in Support of QoS Requirements. *International Workshop on Software Architectures and Mobility (SAM)*. Leipzig, Germany. May 2008.
20. A. Popovici, T. Gross, and G. Alonso. 2002. Dynamic Weaving for Aspect-oriented Programming. *Proceedings of the 1st international Conference on Aspect-Oriented Software Development* (AOSD '02). Enschede, The Netherlands. April 2002. pp. 141-147.
21. Y. Weinsberg, and I. Ben-Shaul. A Programming Model and System Support for Disconnected-Aware Applications on Resource-Constrained Devices. *International Conference on Software Engineering (ICSE 2002).* Orlando, FL. 2002.
22. D. Weyns, H. Parunak, F. Michel, T. Holvoet, and J. Ferber, Environments for multiagent systems, state-of-the-art and research challenges, *Environments for multi-agent systems*. Lecture Notes in Computer Science, vol. 3374, 2005.