Gradual Transition towards Autonomic Software Systems based on High-level Communication Specification

Edin Arnautovic, Hermann Kaindl, Jürgen Falb, Roman Popp and Alexander Szép
Vienna University of Technology
Institute of Computer Technology
Vienna, Austria
{arnautovic, kaindl, falb, popp, szep}@ict.tuwien.ac.at

ABSTRACT
While management of today’s software systems is usually performed by humans using some user interface (UI), autonomic systems would be self-managed. They would typically consist of a managed element, which provides actual system functionality, and an autonomic manager performing system management. However, truly self-managed systems are hard to achieve and not (yet) in wide-spread use. During the transition towards autonomic software systems it is more realistic to manage a large and complex software system partly by humans and partly by an autonomic manager. For facilitating this approach, the communication between the managed element and human administrators on the one hand and the communication between the managed element and the autonomic manager on the other, should be unified and specified on the same semantic level. However, there is no scientific basis for such a unified communication approach.

We present a unified specification of this communication in a high-level discourse model based on insights from theories of human communication. This approach would make this communication “natural” for humans to define and to understand. In addition, we propose to use the same specification for the automated generation of user interfaces for management by human administrators. As a consequence, a smooth and gradual transition towards self-managed software systems will be facilitated, where the portion managed by human administrators becomes smaller and smaller.

Categories and Subject Descriptors

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC’07 March 11-15, 2007, Seoul, Korea
Copyright 2007 ACM 1-59593-480-4/07/0003 …$5.00.

General Terms
Design, Management

Keywords
self-managing systems, autonomic computing, interaction modeling

1. INTRODUCTION
Large and complex software systems need to be managed with respect to, e.g., trouble shooting, dynamic reconfiguration and parameterization. Managing such systems requires dedicated and especially trained personnel and is, therefore, costly. Total costs of computer systems are more and more burdened with the expenses for maintenance, recovery from failures, etc. For example, companies now spend between 33% and 50% of their total cost of ownership recovering from or preparing against failures and 80% of IT money is spent on operations, maintenance and minor enhancements [19]. In addition, a special user interface for these tasks has to be provided, either built into the software to be managed or plugged onto it. Studies show that system administrators work in very complex and challenging environments [1] (both socially and technically). This confirms the need for automation of administration tasks.

In order to address these issues, there is relatively new research on autonomic systems [9] that would be self-managed. The basic idea is to have a so-called autonomic manager, which is itself built in software — as a separate component — and supposed to manage the software system in question. Unfortunately, this ambitious approach is difficult to implement and to put into wide-spread industrial use, especially for legacy systems.

We propose a gradual transition towards self-managed software systems, because it seems to be more realistic in practice. In this approach, “human-managed” and “autonomic-managed” live together, where more and more related tasks will be moved from the human administrator to the autonomic manager. For implementing such an approach, it is desirable to have the communication between the managed software system and its (human or software) manager on the same semantic level. In order to make the communication easy to understand and to specify by humans, the specification of this communication should be on a high level. Unfortunately, the previously proposed protocols for communication between managed elements and autonomic managers are still on a low semantic level (e.g., standard Web services [8]).
Thus, we propose high-level communication specifications based on discourse models to be used during transition towards self-managed software systems. Such discourse models are based on insights from human communication theories and will allow for a unified approach to high-level communication between the managed software system and both its human and its autonomic manager. As a consequence, this communication will be better understandable as well as easier to design by humans. Additionally, the user interfaces can even be generated automatically from such well-defined communication specification.

The remainder of this paper is organized as follows. First, we give a brief summary of the background on autonomic software systems and communication theories used in our approach. Then, we explain our approach to high-level communication between a managed system and its manager in the form of discourses by presenting the discourse metamodel. This metamodel is used for specifying unified communication in our architecture. Finally, we sketch our new transition architecture and conclude.

2. BACKGROUND

2.1 Autonomic Software Systems

Management of software systems includes processes and tasks required to control, measure, optimize and configure software in a computing system. It is crucial to satisfy quality requirements such as performance, availability and security. Nowadays, software systems are managed by humans with possibly some more or less sophisticated tools.

In order to continue to grow in terms of complexity and still being manageable and operating at reasonable costs, software systems should function with least possible human intervention. Humans should ideally only define high-level business policies and the system should adapt to any changes and influences during runtime. A software system should be able to decide on its own what needs to be done in order to keep it stable, secure and fast. Following this, software systems should operate analogously to the human autonomic nervous system, controlling most important body functions without conscious intervention from the brain. Researchers at IBM came up with this idea in 2001, addressing increasing complexity and total cost of ownership [7, 9]. To be autonomic, software systems should have four basic self-managing characteristics [6]: Self-configuration, Self-healing, Self-optimization and Self-protection.

The basic architecture of an autonomic software system consisting of a managed element and an autonomic manager was proposed in [9] and is shown in Figure 1. The managed element implements the main system functionality, and it is controlled and managed by the autonomic manager. We base our work on this architecture but add a novel approach to communication between a managed system and its autonomic manager.

2.2 Communication Theories

Communication with and within software systems can be specified in many ways, where traditionally a major distinction is made whether it is with a human user or within software.

We strive for a unified and high-level approach to communication based on the following work:

![Figure 1: Autonomic Software Systems.](image)

**Communicative acts** Philosophers observed that human speech is also used to do something with intention — to act. Early and seminal work on speech acts was done by Searle [18]. In this essay Searle claims that “speaking a language is performing speech acts, act such as making statements, giving commands, asking questions and so on”. Such speech acts are basic units of language communication. Since speech act theory provides a formal and clean view of communication, computer scientists have found speech acts very useful for describing communication also apart from speech or natural language. To emphasize their general applicability, the notion communicative act is used in this context. Such communicative acts have been successfully used in several applications: inter-agent communication in FIPA Agent Communication Language\(^1\) (ACL), information systems [15] and high-level specifications of user interfaces [5].

**Rhetorical Structure Theory** Rhetorical Structure Theory (RST) [13] is a linguistic theory focusing on the function of text, widely applied to the automated generation of natural language. It describes internal relationships among text portions and associated constraints and effects. The relationships in a text are organized in a tree structure, where the rhetorical relations are associated with non-leaf nodes, and text portions with leaf nodes. In our work we make use of RST for linking communicative acts.

**Conversation Analysis** While communicative acts are useful concepts to account for intention in an isolated utterance, representing the relationship between utterances needs further theoretical devices. We have found inspiration in Conversation Analysis [12] for this purpose. Conversation analysis focuses on sequences of naturally-occurring talk “turns” to detect patterns that are generally specific to human oral communication, and consequently such patterns can be regarded as familiar to the user during human-machine interaction. In our work we make use of patterns such as “adjacency pair”.

3. HIGH-LEVEL DISCOURSE METAMODEL

We model the communication between a managed software system and its (human or autonomic) manager in the form of discourses. Our discourse metamodel is based on insights from several theories of human communication and defines what the models of a communication design specification should look like in our approach. While related concepts have been used for the modeling of human-computer interaction (e.g., in [4]) we use discourse models additionally

---

for communication within software systems, more precisely between a managed software system and its autonomic manager.

We explain our metamodel here using the UML class diagram\(^2\) in Figure 2 and through an example extracted from real systems — a simplified self-optimizing scenario shown in Figure 3. In this scenario, a (human or autonomic) manager of the software system observes server response time (which is too long), determines the cause in the sub-optimal memory usage, and corrects the problem by increasing the heap size. Eventually, the manager checks the server response time once more.

Our new approach to communication models in the form of discourses can be sketched as follows. In essence, it has communicative acts as its “atoms”, from which “molecular” structures can be composed in two dimensions. First, adjacency pairs represent dialogue structure and are modeled in our metamodel by the isAdjacentTo relation. Second, RST relations provide structure relating communicative acts (or adjacency pairs) and further structures made up of RST relations.

The metamodel in Figure 2 consists of two main parts. The left part contains communicative acts, RST relations, and their hierarchical structure; the right part consists of classes involved in the description of management interfaces.

The Communicative Act class carries the intention of the communication like asking a Question about Total CPU Load. The communicative acts can be further classified into Assertions, Directives and Commissives. Assertions convey information without requiring receivers to act beside changing their believes (e.g., Informing and Answer). Directives (e.g., Question, Request, Accept) and Commissives (e.g., Offer) require an action by the receiver or sender and the advancement of the discourse by further communicative acts. This classification is not shown in Figure 2 to avoid cluttering the diagram.

The interactions depicted in the rounded boxes at the bottom of Figure 3 are cast in terms of communicative acts, e.g., the Question for the Total CPU Load shown in a circle with an outgoing arrow. They can be viewed as a usage scenario for optimization which advances from left to right. Since there are many sequences of interactions possible, we could think of this example more generally as a use case. While use cases carry additional information to sequences of actions, they barely represent more complicated structures.

More importantly, neither scenarios nor use cases represent something like intentions of the various interactions. Since our discourse models are built on communicative acts, they specify the type of communicative act for each interaction. E.g., Figure 3 shows Questions and their Responses, as well as Requests with Accepts. This extra piece of information carries the intent of such an interaction.

In addition, according to Conversation Analysis there are frequently occurring pairs of communicative acts — adjacency pairs. E.g., a Question must have a related Response, and a Request must have an Accept or a Reject. Typical adjacency pairs of communicative acts are modeled in our metamodel by the isAdjacentTo relation. In our example, communicative acts which belong together are connected by arrows. The left arrow points from a communicative act to the succeeding one.

As mentioned above, RST relations relate communicative acts and further structures made up of RST relations. The top left part of Figure 2 makes up the hierarchical structure of RST, by generalizing RST relations and communicative acts into nodes which are related by RST relations. In this way, models according to this metamodel form a tree structure. We use two types of RST relations: symmetric (multi-nuclear) and asymmetric (nucleus-satellite) relations.

Multi-nuclear relations like Joint link equal RST structures or communicative acts. In our example, the Joint relation links a question about the Processor 1 Load and a question about the Processor 2 Load. Note, that the order of these actions is not specified by the Joint relation; that is why in this particular example also another scenario would fit in, where, e.g., first Processor 2 and then Processor 1 is asked for its load. If it is possible, these questions could be asked even concurrently.

Nucleus-satellite relations link an RST structure that represents the main intention (nucleus — indicated by ‘N’ in the figure) and an RST structure that supports the nucleus (indicated by ‘S’ in the figure). For example, in Figure 3 the Condition relation requires information about the Server Response Time and the execution continues only if it exceeds...
Figure 3: Self-Optimizing Discourse.

some defined threshold.

The top part of Figure 3 illustrates how all these interactions conceptually belong together as a whole in our example. This structure is composed from RST relations (shown in boxes). For an explanation, let us start at the top with the relation called Result. It represents that the sequence of actions — requesting the increase of the heap size and the request for informing about Server Response Time — is a consequence of the “situation” resulting from the preceding interactions.

In the course of the interactions, some require a certain condition to be fulfilled. E.g., as stated above, the Condition relation requires information about the Server Response Time and the execution continues only if it exceeds some defined threshold. The acquisition of information about the Total CPU load can be further elaborated, adding more details by asking questions about the load of the each particular processor.

Such a tree of RST relations could be viewed as the design rationale of the interactions. Alternatively, it can be viewed as a “plan” structure of the discourse for arriving at some goal. In this view, it is actually a non-linear plan (see the Joint relation in this example), while the usage scenarios are related linear plans. It is important to note that, where the discourse model represents a generic set of possible discourses, the concrete discourse flow will be controlled by the (human or autonomic) manager.

The right part of Figure 2 represents the content of discourse for managing a software system. There are two kinds of information to be exchanged between a software system and its (autonomic or human) manager, namely the information about current system properties and the current action to be executed by the system as requested by the manager. In Figure 3, such a Property is the Total CPU Load and such an Action is IncreaseHeapSize. We define two types of properties: StateInformation and StructuralInformation. StateInformation represent the system as seen from outside using its parameters (black box). StructuralInformation can carry the information about runtime architecture and structure of the system. However, our metamodel does not define the format of the content. So, common standards such as Common Base Event format [16] can be used.

4. TRANSITION ARCHITECTURE

Figure 4 illustrates a sketch of our Self-Managed Communication Platform. It is supposed to facilitate the transition to self-managed systems by supporting unified communication with human administrators and an autonomic manager. During transition towards self-managed operation of the software system, human administrators would be gradually replaced by the software component autonomic manager.

For software systems being manageable, they have to provide two types of interfaces: Sensors and Effectors [8] (S and E in Figure 4). Sensors provide information about current system properties, either on request or on their own initiative. (That is why the related arrow in Figure 4 is bidirectional.) Effectors offer functionality to invoke some action in the system, e.g., for changing some property. (Since these actions are invoked from the outside only, the related arrow
is unidirectional. For example, an application server could provide information about processor and memory load at its sensors and offer functionality for changing parameter values at its effectors.

Today's software systems typically provide such access on the level of object-oriented interfaces or Web services. This involves some form of message passing, metaphorically defined on a slightly higher level than pure procedure calls. Still, such interfaces would be on a lower level than our proposed discourse models based on communicative acts.

Therefore, it will be necessary to "wrap" the lower-level interfaces in a Discourse Engine as indicated in Figure 4. For the autonomic manager as the communication partner, the same kind of "wrapper" has to be provided by another instance of the Discourse Engine. The Discourse Engines implement basically the flow of the discourse. However as stated above, real control of the current discourse has the (human or autonomic) manager. For the human administrator as the communication partner, a UI Generator will have to be provided for automated generation of a user interface. For both communication partners, however, the same communicative acts would flow back and forth in the course of enacting the discourse. For which communicative partner they are translated — human or software, respectively — and how, is to be handled by the Communication Platform.

This architecture covers the whole spectrum of management possibilities. Without an autonomic manager first, it provides for high-level communication with the human administrator only, through a generated user interface. During the transition, the Communication Platform handles the management partly done by the human administrator and partly by the autonomic manager. Over time, more and more management tasks will be taken over by the autonomic manager. Finally, the vision is to have a fully self-managed system. Since the high-level communication between the autonomic manager and the managed system is based on theories of human communication, it would be better understandable and easier to specify by humans.

5. RELATED WORK

Several approaches have been presented for adding the autonomic capabilities on different software system levels. The Accord framework [11] extends and concretizes the basic autonomic architecture by defining three basic interfaces of an autonomic component: functional, control and operational. The functional interface offers standard component functionality with defined inputs and outputs, the control interface offers a set of sensors and effectors, and the operational interfaces offer the possibility to formulate, inject, and manage rules that are used to manage the runtime behaviors of the autonomic element. Rules incorporate high-level guidance in the form of conditional if-then expressions, i.e., "IF condition THEN actions".

In [20] a programming model for effectors is proposed in order to abstract from their technical details. We use the insights from human communication theories in order to make such an abstraction understandable to humans.

User interfaces in the Unity system [3] interact with the underlying system using well-defined programming interfaces. This makes the user interfaces exchangeable, but such programming interfaces are still on a low abstraction level. Our approach abstracts from the type of user interface — both command line and graphical user interfaces could be generated.

Parekh et al. [17] address the addition of autonomic capabilities into legacy systems and thus the transition towards self-managed systems in some sense. They have developed a generic architecture for the interaction between a legacy system to be managed and the autonomic controllers, which perform autonomic system management. The interaction is based on publish/subscribe event notification and is still on a low abstraction level.

In general, many researchers work on making the systems more self-managed. However, the field lacks generic approaches [10]. This is in particular the case for the problem of the transition towards autonomic systems. Our approach should fill this gap. We are not aware of any complete, defined approach for the transition from human- to self-managed software systems.

6. CONCLUSION AND FUTURE WORK

In this paper, we present a high-level discourse metamodel to be used for unified communication during the transition towards autonomic software systems. For the utilization of these concepts we also present a new transition architecture. This will facilitate and hopefully speed up the transi-
tion from human- to self-managed software systems. Since this transition is challenging and should happen gradually, foundations and tools — as presented in this paper — are crucial. Additionally, our communication model can be used for communication modeling within fully autonomic systems where this communication would be easier to understand and design by humans.

We are currently working on a prototypical implementation of our approach on top of the Java application server (JBoss3). We will evaluate a transition according to our architecture and the resulting autonomic capabilities (see, e.g., [2]). Future research will include further development of our architecture and its evaluation as well as validation of our discourse metamodel against self-management scenarios. Another part of the research will investigate a transition process using our approach according to the IBM autonomic maturity levels criteria [14].

7. REFERENCES


3www.jboss.org