On Decentralized Self-Adaptation: Lessons from the Trenches and Challenges for the Future

Danny Weyns
Dept. of Computer Science
Katholieke Universiteit Leuven
danny.weyns@cs.kuleuven.be

Sam Malek
Dept. of Computer Science
George Mason University
smalek@gmu.edu

Jesper Andersson
Dept. of Computer Science
Linnaeus University
jesper.andersson@lnu.se

ABSTRACT

Self-adaptability has been proposed as an effective approach to deal with the increasing complexity, distribution, and dynamicity of modern software systems. Although noteworthy successes have been achieved in many fronts, there is a lack of understanding on how to engineer distributed self-adaptive software systems in which central control is not possible. In this paper, we first describe the key attributes of decentralized self-adaptive systems that set them apart from their centralized counterparts. We illustrate these attributes using two case studies on decentralized self-adaptation. The first case study is an instance of a self-healing system dealing with automated traffic management control. The second case study is an instance of a self-optimizing system that improves the quality of service of a decentralized software system through redeployment of its software components. We generalize the lessons learned from our experiences in the form of a reference model. In light of this model, we present numerous challenges that forms the focus of future research in this area.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*

General Terms

Design

Keywords

Self-adaptation, decentralized control

1. INTRODUCTION

Due to the increasing complexity, distribution, and dynamicity of software systems, assuring and maintaining the required qualities of software constitutes a tremendous challenge. Self-adaptability has been proposed as an effective approach to tackle the increasing complexity of managing modern-day software systems after their initial deployment. Self-adaptability endows a system with the capability to adapt itself to changes in its environment and user requirements.

For managing the complexity of dynamic adaptation several solutions have been proposed. Researchers have previously argued that software architecture provides the right level of abstraction and generality to deal with the challenges of self-adaptability [11, 19, 29]. The adaptation process itself is supported by various frameworks [11, 29, 17], which exploit principles and techniques from feedback control loops [33] and computational reflection [20] to manage system adaptation. The adaptation process monitors the system and typically uses an architecture representation of the system to perform an adaptation when the conditions for that adaptation hold. Existing approaches typically support centralized or hierarchical control of adaptation.

Although noteworthy successes have been achieved in many domains, there is a lack of understanding on how to engineer distributed self-adaptive software systems in which central control is not possible [2, 4]. Decentralized control is crucial for quality requirements such as resilience, robustness, and scalability. In a large class of modern distributed systems, such as web-scale information systems, intelligent transportation systems, and power grids, global control is difficult to achieve or even infeasible, although centralized control of local subsystems is possible.

In this paper, we first lay down the key capabilities required for engineering a decentralized self-adaptive software system. In particular, through careful study of literature and reflecting on our own experiences with the development of such systems, we have distilled several key attributes that distinguish these systems from the more commonly found centralized self-adaptive systems.

Afterwards, we illustrate these attributes using two case studies. In the first case study, intelligent cameras collaborate to detect and monitor traffic jams on the highway in a decentralized way, avoiding the bottleneck of a centralized control center. Our particular interest in this case is on robustness to camera failures. Therefore, cameras are equipped with support for self-healing that allows the system to detect failures and recover autonomously. In the second case, we describe a decentralized instantiation of a framework intended to improve a software system's quality of service via redeployment of its components.

We generalize the lessons we have learned in the development of these systems in the form of a high-level reference model. The reference model reconceptualizes constructs and principles found in existing commonly known frameworks to fit the unique requirements of decentralized self-adaptation. In light of the reference model and our experiences we identify a number of key challenges that form the focus of future research in this area.

The remainder of this paper is structured as follows. Section 2 outlines the required capabilities for engineering decentralized self-adaptive systems. Section 3 presents the first case study on self-healing in the traffic monitoring application. Section 4 presents the second case study on a self-optimization framework. Section 5 generalizes from the case studies by presenting a reference model. Finally, Section 6 draws conclusions and formulates a number of challenges for future research in this area.

2. MOTIVATION

The underlying motivation in our work has been the lack of adequate techniques and principles for systematic engineering of decentralized self-adaptive software systems. In this section, we elaborate on this assertion by first zooming in on the unique characteristics of decentralized self-adaptation. These characteristics in turn enable us to explain the shortcomings of existing frameworks, the majority of which make certain assumptions on the availability of information and flow of control that are not suitable in this setting.

2.1 Decentralized Self-Adaptation

A self-adaptive software system is typically developed in a layered architecture. As further detailed in Section 5, the reference model presented in this paper builds on our previous work [1], in which we have demonstrated the prominent role of reflection in self-adaptive software, and in particular the fact that through reflection one could clearly separate concerns and identify its layers. The bottom layer comprises the software that deals with the application logic which we call the *managed system* (a.k.a, base-level subsystem). On top of that, the self-adaptive layer is added that comprises the software that deals with a particular concern (e.g., performance, robustness, etc) which we call the *self-adaptive unit* (a.k.a., meta-level subsystem). The self-adaptive unit comprises components that monitor the managed system and based on a set of goals adapt it to changing conditions. Additional layers with self-adaptive units could be added that deal with higher level concerns and goals.

A decentralized self-adaptive software system is a special form of autonomic system in which there is no central point of control. The lack of a central point of control could manifest itself in various stages of autonomic computing (i.e., monitoring, analysis, planning, and execution). Moreover, the lack of a central point of control often implies the unavailability of global knowledge on any particular host. A self-adaptive layer in such systems thus consists of several self-adaptive units with the corresponding locally managed systems. As a consequence, self-adaptive units that deal with a concern that involves multiple local managed systems have to collaborate.

Based on a thorough study of literature (e.g., [17, 19, 29, 5, 10, 9]) and reflecting on our own experiences, which are further discussed in Sections 3 and 4, we identified a set of required capabilities for such systems that differentiate them from the more commonly found centralized self-adaptive systems. We distinguish between requirements for the different types of computations in self-adaptive systems.

Coordinated monitoring collects data of the underlying managed system and possibly of the operating environment in which the system is situated (e.g., hardware devices, network connections). In a centralized setting, the self-adaptive unit has access to the entire managed system. As such, the self-adaptive unit has global knowledge of the underlying system. In a decentralized setting, on the other hand, each self-adaptive unit has only access to the local managed system. Consequently, it has only a partial model of the complete system. Therefore, monitor computations of different self-adaptive units may need to coordinate to share and synchronize locally collected data.

Coordinated analysis interprets the collected data to determine goal violations and predicts possible future situations. In a centralized setting, analyze computation has all the data required to perform the analysis, while in decentralized self-adaptive systems, analyze computation often has only partial knowledge. As a consequence, analysis computations of different self-adaptive units may need to coordinate with one another, such that adaptation options are evaluated within a group.

Coordinated planning constructs the actions needed to achieve the system's objectives. In a centralized setting, the self-adaptive unit has access to the goals of the self-adaptive system. As such, plan computation may select and construct an adaptation plan that satisfies the system's objectives. In a decentralized setting, self-adaptive units may have private goals (i.e., be selfish). Moreover, there may be conflicts between the goals of different units. As such, self-adaptive units exhibit autonomous behavior. Consequently, plan computations may need to coordinate (e.g., negotiate) with one another to construct an adaptation plan that satisfies the different goals.

Coordinated execution carries out the changes to the underlying system. In a centralized setting, the self-adaptive unit has full control over the order and timing in which adaptations are executed. In a decentralized setting, execute computations may need to synchronize the changes to the system. In particular, executing transactional adaptation poses a challenge in a decentralized setting by severely disrupting the system.

2.2 Existing Frameworks for Self-Adaptation

Since the late 1990s multiple perspectives on self-adaptive and autonomic software systems have been developed [29, 9, 17, 19]. Several of these works provide high-level reference models and reference frameworks that constitute a solid ground for self-adaptive systems. Key principles that underlie these approaches are the control-loop, as expressed in early work of Shaw [33], and more recently, Dobson et al. [8] and Brun et al. [3], and computational reflection, Maes [20], which was extended further to software architecture by Tisato et al. [35], and to reflective middleware by Coulson et al. [6]. One of the rare proposals that considers decentralization at the outset is [12]. In this work, software components automatically configure themselves according to an overall architectural specification without central configuration management service. The presented approach uses reliable broadcast channels to maintain local copies of the configuration and coordinate component managers. As mentioned by the authors, this restrict the scalability of the approach. The question of how to solve this problem is left open by the authors.

In light of the attributes identified earlier, we take a close look at two representative seminal frameworks for the construction of self-adaptive software systems: Rainbow [11] and IBM's autonomic manger [17]. While we believe these framework have achieved noteworthy success in alleviating challenges of developing self-adaptive software in centralized and hierarchical settings, they are neither directly suitable for decentralized self-adaptation, nor that has been their intention.

The Rainbow framework offers a generic architecture for building self-adaptive systems. The architecture of Rainbow consists of three layers: *system* layer, *translation* layer, and *architectural* layer. The system layer provides the application logic. It offers probes for measuring properties of interest, such as the workload of servers, and effectors for performing changes. The translation layer is responsible for bridging the abstraction gap between the system layer and the architectural layer. The architectural layer deals with self-adaptation. This layer includes an architectural model of the executing system consisting of components, connectors, properties associated with components and connectors, and constraints that define allowed configurations. A constraint evaluator periodically evaluates the constraints of the architectural model and in case of constraint violations triggers an adaptation of the system.

The Rainbow framework supports monitoring and adaptation of software systems that are distributed in a network. However, the control of adaptation is centralized. Rainbow does not provide explicit support for adaptation scenarios where two or more adaptation control units have to collaborate to achieve self-adaptation.

IBM's approach of autonomic systems proposes a layered architecture. The bottom layer consists of system components which could be either software or hardware resources. Software resources correspond to the application logic. Each resource is managed by an autonomic manager. Autonomic managers are the basic building blocks for realizing self-adaptation in autonomic systems. The autonomic manager provides a traditional control loop consisting of four basic computations: monitor, analyze, plan, and execute. Four concrete types of autonomic managers are distinguished: managers for self-configuring, self-optimizing, self-healing, and self-protecting. The autonomic manager itself provides sensors and actuators that enables other autonomic managers to manage the autonomic manager. This enables hierarchical composition of autonomic managers. E.g., a set of resource autonomic managers that deal with self-healing may be managed by an orchestrating autonomic manager. In a hierarchy of autonomic managers, data can be obtained and shared via knowledge sources, such as a registry, dictionary, or a database.

IBM's autonomic systems support hierarchical arrangement of autonomic managers. These managers can share information via knowledge sources. However, autonomic managers do not provide explicit support to interact as peers in a decentralized architecture. Autonomic managers in the bottom layer of the hierarchy have adaptation goals specific to the concern of interest to the managed resource, while autonomic managers in the higher layers deliver system-wide autonomic capabilities.

As illustrated above, and further corroborated by our study of other approaches (e.g., [9, 29]), existing frameworks for self-adaptive systems are geared to centralized/hierarchical systems and do not really consider the capabilities required for building decentralized solutions. This has been the key motivation for our work.

3. CASE STUDY ON SELF-HEALING

The first case study presents a traffic monitoring application. We start the section with introducing the application. Then we explain the decentralized self-healing approach for dealing with camera failures.

3.1 Traffic Monitoring Application

The monitoring application we consider fits in the domain of intelligent transportation systems, a worldwide initiative to exploit information and communication technology to improve traffic. The system consists of a set of intelligent cameras which are distributed evenly along the road. An example of a highway is shown in Figure 1. Each camera has a limited viewing range and cameras are placed to get an optimal coverage of the highway with a minimum overlap.

Cameras are equipped with a data processing unit capable of processing the monitored data, and a communication unit to communicate with other cameras. A camera is able to measure the local traffic conditions and decide whether there is a traffic jam. The task of the cameras is to detect and monitor traffic jams on the highway in a decentralized way, avoiding the bottleneck of a centralized control center. Possible clients of the monitoring system are traffic light controllers, driver assistance systems such as systems that inform drivers about expected travel time delays, systems for collecting data for long term structural decision making, etc.

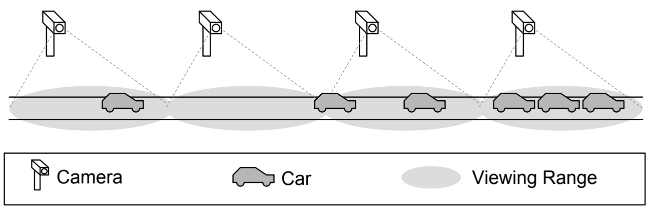

Figure 1: An example of a highway with traffic cameras.

Traffic jams can span the viewing range of multiple cameras and can dynamically grow and dissolve. By default each camera monitors the traffic state within its viewing range. When a traffic jam occurs, the camera has to collaborate with other cameras detecting the same traffic jam. Because there is no central point of control, cameras have to aggregate the monitored data to determine the position of the traffic jam on the basis of the head and tail of it. One of the cameras is responsible to distribute the aggregated data of the traffic jam to the interested clients. Cameras enter or leave the collaboration whenever the traffic jam enters or leaves their viewing range.

In this paper we are interested in one of the quality concerns of the traffic monitoring system: robustness to camera failures. Our particular focus is on silent node failures, i.e., failures in which a failing camera becomes unresponsive without sending any incorrect data. Such failures may bring the system to an inconsistent state and disrupt its services.

3.2 Managed System

We start with a high-level overview of the software architecture of the system that deals with the domain functionality. Then we explain how this architecture is extended with support for self-healing. In particular, we focus on the support for its decentralization.

Figure 2 shows the primary elements of the *main system* which provides the camera software that deals with the domain functionality. The main system is conceived as an agent-based system consisting of four layers. The *host infrastructure* 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, including support for interaction with the environment and communication. The *organization middleware* layer encapsulates the life-cycle management of organizations and it provides role-specific services to the agents to interact with the environment and communicate with other agents. The agents in the *agent layer* use the organization middleware to interact with each other through the roles they play in the organizations, providing the services to the clients of the system, i.e., reporting traffic jams.

We briefly explain how the agents collaborate to provide the system services. In normal traffic conditions, each *camera agent* belongs to a *single member organization* and plays the *role* of *data observer*, monitoring the local traffic. We say that the agent has a *role contract* with the organization. A role contract is a mutual agreement between the agent and the organization that allows the agent to play the role in that organization. In the role of data observer, the agent shares the traffic conditions it monitors with the organization. When the traffic conditions change and a jam is detected, the organization middleware opens a *role position* for the *data pusher* role. A role position can be considered as a vacancy for a particular role in a particular organization. When the agent accepts this role position it gets a role contract. In the role of data pusher, the agent is responsible for collecting traffic data in the organization and notifying interested clients of the traffic jam. The organization middleware stores the role contracts with the local agent and advertises the open role positions. When the traffic jam grows, the organization middleware joins the neighboring organizations in a single organization that spans the viewing range of the cameras that monitor the traffic jam. In this joined organization, only one agent plays the role of data pusher. When the traffic jam resolves, the organization is split dynamically.

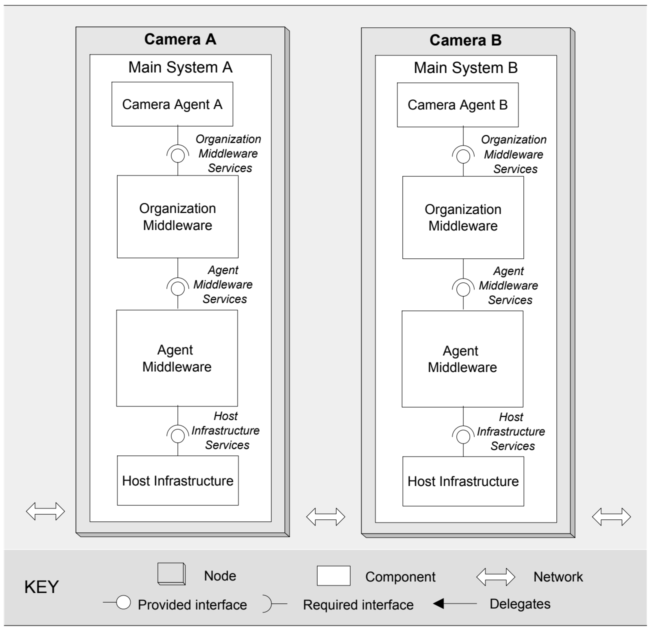

Figure 2: Main system deployed on each camera.

Organization dynamics are managed by *organization controllers*. Organization controllers need to collaborate to manage an organization that spans multiple nodes. Neighboring organizations that are candidates for merging are denoted as *interaction candidates*. To make the system scalable, a decentralized peer-to-peer approach is used to manage the inter-organizational dynamics. However, to simplify synchronization issues, a master/slave principle is used at intra-organizational level. The master/slave principle is a control model, which centralizes (locally) the control of each organization and the state required for this control. For each organization, one of the organization controllers is elected as master, whereas the other controllers of the organization are slaves. The master is responsible for managing the dynamics of that organization by synchronizing with all of the slaves.

3.3 Decentralized Self-Healing

To support robustness to node failures, a *self-healing subsystem* is added to the main system that is responsible for dealing with camera failures. The self-healing subsystem is a domain specific instance of a *self-adaptive unit*, while the *main system* corresponds to a *local managed system*.

Figure 3 shows a deployment view of one node of the main system extended with the self-healing subsystem. A self-healing subsystem comprises the following components:

- *Dependency Model.* The dependency model contains a model of the dependencies of the components of the main system with other cameras. The *Query/Update* interface provides access for inspecting and updating the model. Figure 4 shows an overview of the dependencies in the system.

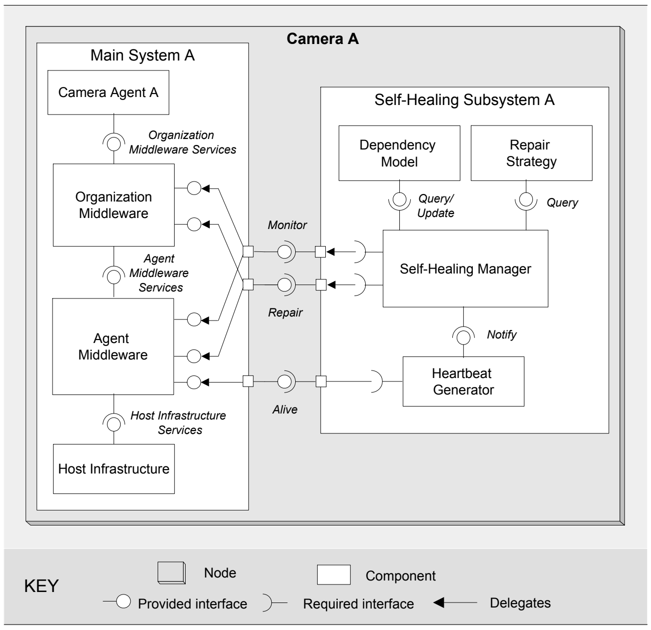

Figure 3: Deployment view with self-healing subsystem.

- *Heartbeat Generator.* The heartbeat generator periodically receives alive messages from all of the nodes (cameras) that it depends on, and sends alive messages to all of the nodes that depend on its node. To exchange messages, the heartbeat generator uses the *Alive* interface provided by the main system, which exploits the communication facilities of the agent middleware. The *Notify* interface notifies the *self-healing manager* whenever an alive message arrives from a camera with a dependency in the dependency model.

- *Repair Strategy.* The repair strategy component contains a set of *repair actions* to bring the main system to a consistent state in case of a failure, i.e., when a camera this node depends on fails and no longer sends alive messages. Each set of repair actions represent a plan to deal with a particular failure. Figure 4 provides an overview of the repair actions employed in the system.

- *Self-Healing Manager.* The self-healing manager component contains the logic to deal with self-healing. The self-healing manager monitors the main system using the *Monitor* interface to maintain the dependency model. It interprets the notifications of the *heartbeat generator* to derive the status of the cameras with a dependency in the dependency model. Finally, when a failure is detected, the self-healing manager executes the repair actions of the repair strategy using the *Repair* interface to bring the main system to a consistent state.

Self-healing subsystems monitor the main systems for failures and only interfere in their operation when a failure is detected. The self-healing subsystems ensure the system can continue its correct operation (possibly in a degraded mode, e.g., one camera less) for a limited time after a failure occurs.

The self-healing manager encapsulates the typical computations of a self-adaptive unit: monitor, analyze, plan, and execute. The monitor computation gathers the required data of the main system locally. However, it requires coordination with self-healing managers to gather the status of the nodes in the dependency model. This coordination is realized using the heartbeat mechanism.

	Dependencies	Nodes to monitor	Repair actions
Any Node	- adjacent cameras	- all neighboring nodes	- remove neighbor, calculate and set new neighbor
	- communication links	- all linked nodes	- remove communication link in agent middleware
Slave Nodes	- master of organization	- master node of organization	- start new single-member organization
	- role contracts	- set of other slave nodes	- remove role contract from local repository
Master Nodes	- slaves of organization	- all slave nodes	- remove slave reference in organization controller
	- open role positions	- set of slave nodes	- remove role position from local repository
	- role contracts	- set of slave nodes	- remove role contract from local repository
	- interaction candidates (neighbor organizations)	- set of other master nodes	- remove interaction candidate from local organization context

Figure 4: Dependencies with corresponding nodes to monitor and repair actions.

To recover from a silent node failure, the self-healing subsystems have only to apply local repair actions. The required data for analyzing, planning, and executing is available locally and the self-healing subsystems engage in local repair actions when a failure is detected. In other words, there are no negotiations between subsystems to analyze a failure or execute repair actions. The self-healing subsystem brings the main system back to a consistent state, from which it can continue its correct operation on its own. Support for more advanced repair actions, such as electing a new master when the master fails, would require the coordination among the plan and execute computations of different self-healing subsystems.

The decentralized approach for self-healing described above builds upon the MACODO model and middleware platform. Interested reader may refer to [36].

4. CASE STUDY ON QOS-DRIVEN SELF-OPTIMIZATION

For any large, distributed system many deployment architectures (i.e., distributions of the system's software components onto its hardware hosts) are typically possible. However, some of those deployment architectures are more dependable than others. For example, a distributed system's availability can be improved if the system is deployed such that the most critical, frequent, and voluminous interactions occur either locally or over reliable and capacious network links.

Finding a deployment architecture that exhibits desirable system characteristics (e.g., low latency, high availability) or satisfies a given set of constraints (e.g., the processing requirements of components deployed onto a host do not exceed that host's CPU capacity) is a challenging problem: many system parameters (e.g. network bandwidth, reliability, frequencies of component interactions, etc.) that influence the selection of an appropriate deployment architecture are typically not known at system design time and/or may fluctuate at run time; the space of possible deployment architectures is extremely large, thus finding the optimal deployment is rarely feasible [21]; and different desired system characteristics may be conflicting (e.g., a deployment architecture that satisfies a given set of constraints and results in specific availability may at the same time exhibit high latency).

The above problem is further complicated in the context of the emerging class of decentralized systems, which are characterized by limited system-wide knowledge and the absence of a single point of control. In decentralized systems, selection of a globally appropriate deployment architecture has to be made using incomplete, locally-maintained information.

A methodology for improving a distributed system's quality of service properties consists of (1) active system monitoring, (2) estimation of the improved deployment architecture, and (3) redeployment of (parts of) the system to effect the improved deployment architecture. Based on this three-step methodology, a high-level deployment improvement framework [22, 21] has been developed. In this section, we provide an overview of the framework's components, the associated functionality of each component, and the dependency relationships that guide their interaction. In particular, we focus on the techniques employed for decentralization of the approach.

4.1 Deployment Improvement Framework

Figure 5 shows the framework's overall structure and the relationships among its six high-level components. Note that each of the framework's components can have an internal architecture that is composed of one or more lower-level components. Furthermore, the internal architecture of each component can be distributed (i.e., different internal low-level components may communicate across address spaces). The arrows represent the flow of data among the framework components. The *deployment improvement framework* depicted in Figure 5 corresponds to a *self-adaptive unit* that is deployed on each host. The framework manages a software system running on top an *implementation platform*, which together correspond to a *local managed system*.

Model. This component maintains the representation of the system's deployment architecture. The model is composed of four types of parts: hosts, components, physical links between hosts, and logical links between components. Each of these types could be associated with an arbitrary set of parameters. For example, each host can be characterized by the amount of available memory, processing speed, battery power (in case a mobile device is used), installed software, and so on. The selection of a set of parameters to be modeled depends on the set of criteria (i.e., objectives) that a system's deployment architecture should satisfy. For example, if minimizing latency is one of the objectives, the model should include parameters such as physical network link delays and bandwidth. However, if the objective is to improve a distributed system's security, other parameters, such as security of each network link, need to be modeled.

Algorithm. Each objective is formally specified and can either be an optimization problem (e.g., maximize availability, minimize latency) or constraint satisfaction problem (e.g., total memory of components deployed onto a host cannot exceed that host's available memory). Given an objective and the relevant subset of the system's model, an algorithm searches for a deployment architecture that satisfies the objective. An algorithm may also search for a deployment architecture that simultaneously satisfies multiple objectives (e.g., maximize availability while satisfying the memory constraints). In terms of precision and computational complexity, there are two categories of algorithms for an optimization problem like this: exact and approximative. Exact algorithms produce optimal results (e.g., deployments with minimal overall latency), but are exponentially complex, which limits their applicability to systems with very small numbers of components and hosts. On the other hand, approximative algorithms in general produce sub-optimal solutions, but have polynomial time complexity, which makes them more usable.

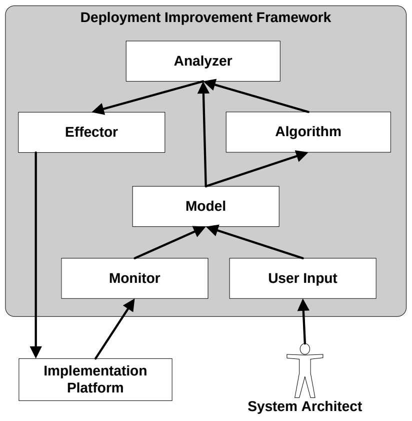

Figure 5: Deployment improvement framework overview.

Analyzer. Analyzers are meta-level algorithms that leverage the results obtained from the algorithm(s) and the model to determine a course of action for satisfying the system's overall objective. In situations where several objective functions need to be satisfied, an analyzer resolves the results from the corresponding algorithms to determine the best deployment architecture. However, note that an analyzer cannot always guarantee satisfaction of all the objectives. Analyzers are also capable of modifying the framework's behavior by adding or removing low-level components from the framework's high-level components. For example, once an analyzer determines that the system's parameters have changed significantly, it may choose to add a new low-level algorithm component that computes better results for the new operational scenario. Analyzers may also hold the history of the system's execution by logging fluctuations of the desired objectives and the parameters of interest. System's execution profile allows the analyzer to fine-tune the framework's behavior by providing information such as system's stability, work load patterns, and the results of previous redeployments.

Monitor. To determine the run time values of the parameters in the model, a monitor is associated with each parameter. The monitor is implemented in two parts: a platform-dependent part that "hooks" into the implementation platform and performs the actual monitoring of the system, and a platform-independent part that interprets and may look for patterns in the monitored data. For example, it determines if the data is stable enough to be passed on to the model.

Effector. Just like monitors, effectors are also composed of two parts: (1) a platform-dependent part that "hooks" into the platform to perform the redeployment of software components; and (2) a platform-independent part that receives the redeployment instructions from the analyzer and coordinates the redeployment process. Depending on the implementation platform's support for redeployment, effectors may also need to perform tasks such as buffering, hoarding, or relaying of the exchanged events during component redeployment.

User Input. Some system parameters may not be easily monitored (e.g., security of a network link). Also, some parameters may be stable throughout the system's execution (e.g., CPU speed on a given host). The values for such parameters are provided by the system's architect at design time. We are assuming that the architect is able to provide a reasonable bound on the values of system parameters that cannot easily be monitored. Furthermore, the architect also must be capable of providing constraints on the allowable deployment architectures. Examples of these types of constraints are location and collocation constraints. Location constraints specify a subset of hosts on which a given component may be legally deployed. Collocation constraints specify a subset of components that either must be or may not be deployed on the same host.

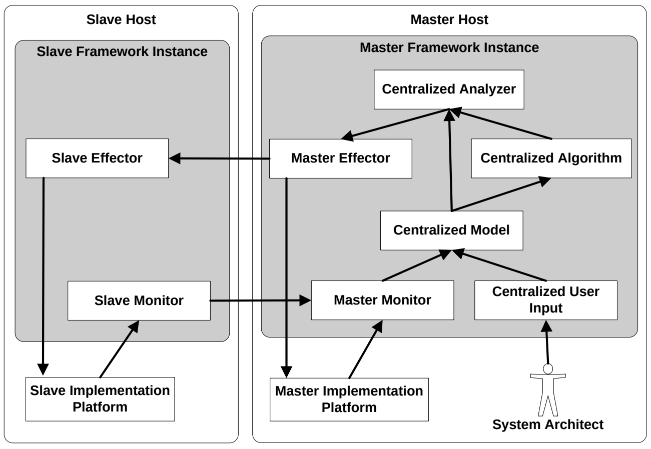

Figure 6: Deployment improvement framework distributed centralized instantiation.

The framework described above has been realized using an integration of several tools, including Prism-MW [23], an architectural middleware platform with monitoring and component migration capabilities, and DeSi [26], a software deployment modeling and analysis environment. For the sake of brevity the details of these tools are not provided. Interested reader may refer to [22, 23, 26].

4.2 Decentralization of the Framework

Figure 6 shows the framework's instantiation in a distributed centralized setting. Centralized systems have a *master host* (i.e., central host) that has complete knowledge of the distributed system parameters. Master host contains a *centralized model*, which maintains the global model of the distributed system. The centralized model is populated by the data it receives from *master monitor* and *centralized user input*. The master monitor receives all of the monitoring data from the *slave monitors* on other hosts. Once all of the monitoring data from all of the slave hosts is received, the master monitor forwards the monitoring data to the centralized model. Each slave host contains a *slave effector*, which receives redeployment instructions from the *master effector*, and a slave monitor, which monitors the slave host's implementation platform and sends the monitoring data back to the master monitor. Finally, the master effector receives a sequence of command instructions from the *centralized analyzer* and distributes the redeployment commands to all the slave effectors.

Figure 7 shows the framework's instantiation for a decentralized system. Unlike a centralized software system, a decentralized system does not have a single host with the global knowledge of system parameters. Each host has a *local monitor* and a *local effector* that are only responsible for the monitoring and redeployment of the host on which they are located. Each host has a *decentralized model* that contains some subset of the system's overall model, populated by the data received from the local monitor and the decentralized model of the hosts to which this host is connected. Therefore, if there are two hosts in the system that are not aware of (i.e., connected to) each other, then the respective models maintained by the two hosts do not contain each other's system parameters. Each host also has a *decentralized algorithm* (e.g., auctioning [21]) that synchronizes with its remote counterparts to find a common solu-

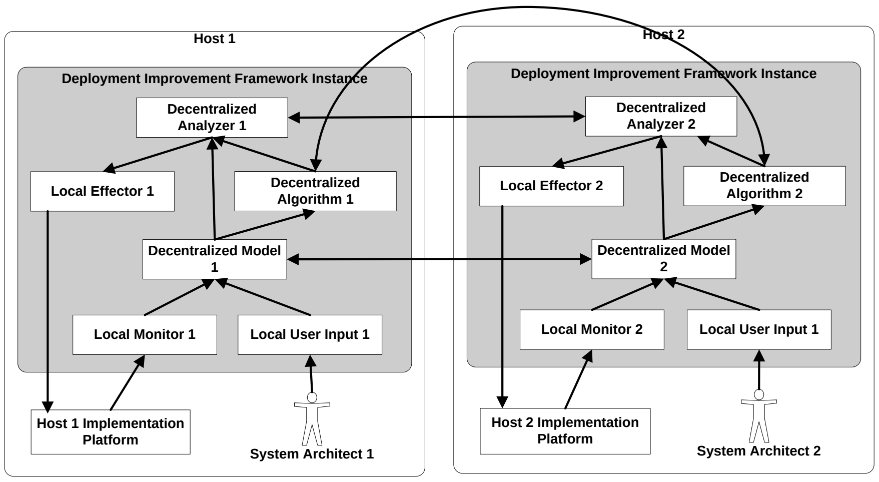

Figure 7: Deployment improvement framework decentralized instantiation.

tion. Finally, in a similar way, the *decentralized analyzer* on each host synchronizes with its remote counterparts to determine an improved deployment architecture and effect it.

5. REFERENCE MODEL FOR DECENTRALIZED SELF-ADAPTATION

Reflecting on our experiences with the two case studies has helped us to gain a better understanding of decentralized self-adaptive systems. In particular, it has illuminated several necessary extensions to the commonly employed principles and frameworks. We generalize the lessons we have learned in terms of a widely applicable reference model depicted in Figure 8. This model brings together concepts from computational reflection [20], MAPE [17], and feedback-control loop [33], and extends them with the additional constructs necessary for decentralization.

A *self-adaptive system* comprises one or more *local self-adaptive systems*. For instance, the traffic monitoring system is an example of a self-adaptive system. The software deployed on a camera is a local self-adaptive system. A local self-adaptive system consists of a set of *local managed systems* and a set of *self-adaptive units*. For example, the main system in the traffic monitoring case and host implementation platform in the deployment improvement system are examples of a local managed system. On the other hand, self-healing subsystem and deployment improvement framework software are examples of a self-adaptive unit.

A self-adaptive unit is a self-contained entity that adapts the local managed system using several *meta-level computations*, which use a set of *meta-level models*. Below we explain meta-level models and meta-level computations in detail.

5.1 Meta-Level Models

We distinguish between four types of *meta-level models*: *system model*, *concern model*, *working model*, and *coordination model*.

The *system model* represents (parts of) the *system* that is managed by the self-adaptive unit. The system can be either a local managed system or a self-adaptive unit. The latter is applicable to self-adaptive units that deal with higher level concerns (i.e., a meta-metal-level model). The decentralized model in the deployment improvement system that maintains a representation of the local system's deployment architecture is an example of a system model. In the traffic monitoring case, the dependency model contains data that corresponds to a system model, i.e., the elements that represent the components of the main system and the cameras on which those components depend.

The *concern model* represents the objectives (goals) of a self-adaptive unit. The concern model in the deployment improvement system is represented either as an optimization problem (e.g., maximize availability), or a constraint satisfaction problem (e.g., keep components' memory usage within the boundaries of the host's available memory). In the traffic monitoring system, the self-healing concern is represented as rules of the form *event–condition–action set*. *Event* is a failure of a camera, *condition* is a local dependency on the failing camera, and *action set* are the set of repair actions required to recover from the failure. The self-healing manager uses data from the dependency model and the repair strategy to derive the *event–condition–action set* rules.

A *working model* represents the data structures and information shared between the meta-level (MAPE) computations. These models are typically domain-specific. Examples of working models in the deployment improvement system are the temporary representations of candidate deployment architectures that are evaluated by an analyzer, and the redeployment instructions produced by the analyzer and used by the effector (see Figure 7).

A *coordination model* is a key model in decentralized self-adaptive systems that distinguishes them from their centralized counterparts. A *coordination model* represents the data required by a meta-level computation to coordinate with meta-level computations of other self-adaptive units. For example, in the simple scenario shown in Figure 7, each of the decentralized analyzers, decentralized models, and decentralized algorithms maintains a representation of its peers. Moreover, the coordination model may contain data about ongoing negotiations. For instance, a decentralized analyzers from Figure 7 may use an auction mechanism to find an agreeable deployment solution with its counterparts, and the status of the ongoing interaction is captured in the coordination model. In the traffic monitoring case, each self-healing manager maintains a representation of the cameras that its node has a dependency on, as specified in the dependency model. These dependencies mimic the organizational structure of the agents that deal with the domain functionality. In general, the data maintained in a coordination model may range from a set of simple references to peer computations to a complex organization model that computations use to coordinate. We further elaborate on coordination activity below.

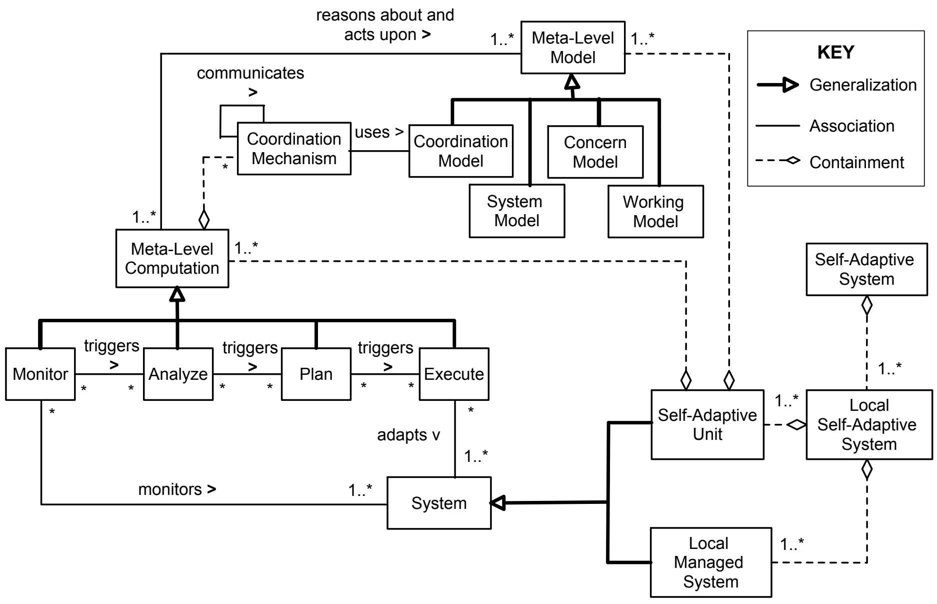

Figure 8: Reference model for decentralized self-adaptive systems.

5.2 Meta-Level Computations

Meta-level computations are the typical feedback control loop computations found in self-adaptive systems–*monitor*, *analyze*, *plan*, and *execute*–enhanced with support for decentralized coordination. The reference model explicitly separates coordination from computation. Coordination is dealt with by *coordination mechanisms* that use *coordination models*.

In general, a *coordination mechanism* allows resolution of coordination problems arisen from dependencies [24]. Coordination mechanisms can be as simple as a priority-based synchronization mechanism to a complex market mechanism. [7] proposes a basic taxonomy of coordination mechanisms based on the types of dependencies that underlie coordination problems (e.g., managing dependencies between multiple tasks and multiple resources). In decentralized self-adaptive systems, the choice for a particular coordination mechanism depends on the type of computations that have to coordinate and the characteristics and requirements of the domain at hand. Below we provide concrete examples of coordination mechanisms for the different types of meta-level computations.

Monitor computation monitors the *system* that is managed by the self-adaptive unit and possibly aspects of the system's environment. The system can be either a local managed system or another lower-level self-adaptive unit. Monitor uses the observed data to update the system model and may *trigger* analyze computations when particular conditions hold. In the traffic monitoring case, the self-healing manager monitors the main system to keep the dependency model up to date. To keep track of the status of remote cameras with dependencies, each self-healing manager uses a *heartbeat generator* that broadcasts alive messages. Heartbeat is the coordination mechanism used by self-healing managers to realize coordinated monitoring. In the deployment improvement system, the local monitor on each host keeps track of the managed system via the local implementation platform's probes and gauges, and populates the corresponding decentralized model. In addition, the decentralized models that are directly connected to one another (e.g., within close proximity) realize coordinated monitoring by exchanging data. Deployment models may share data using a variety of coordination mechanisms, such as gossiping and proximity broadcasting.

Analyze computation assesses the collected data to determine the system's ability to satisfy its objectives. *Plan* computation constructs the actions necessary to achieve the system's objectives. Analyze computations may *trigger* plan computations, e.g., when a particular analysis determines violation of system's objectives. Analysis and planning in the traffic monitoring case is trivial: the self-healing manager checks the alive messages of cameras with dependencies and when a camera fails, it triggers the repair actions to adapt the main system. Since only local adaptations are needed, there is no need for coordinated analysis or planning. In the deployment improvement system, a decentralized algorithm uses different coordination mechanisms (e.g., auctioning, voting) to analyze possible reconfigurations. Decentralized analyzer performs coordinated planning; i.e., analyzers of connected hosts synchronize to determine an improved deployment architecture for the managed system.

Finally, triggered by plan, *execute* computation carries out changes on the managed system. In the traffic monitoring case, the self-healing manager executes the repair actions to bring the main system to a consistent state. In the deployment improvement system, the local effectors deal with the low-level migration and installation of software components via the facilities provided by the implementation platform. While not explicitly depicted in Figure 7, the effectors coordinate with one another by requesting migration of the required software components (i.e., as specified in the new configuration selected by decentralized analyzer) from their peers.

6. CONCLUSION AND CHALLENGES AHEAD

Over the last decade, research on self-adaptive systems has matured to encompass a set of foundational principles, techniques, and implementation frameworks. Self-adaptive systems have been built for a variety of concerns in different domains. Still, state-of-the-art self-adaptive frameworks lack support for a growing class of systems in which central control is not an option. Based on a thorough study of existing approaches and reflecting on our own experiences, we derived a set of key capabilities required for engineering decentralized self-adaptive systems, and illustrated them in two cases studies.

We generalized the lessons learned in the fyorm of a high-level reference model. This model reifies the basic constructs of self-adaptive systems and extends them with additional constructs to support the engineering of decentralized self-adaptive systems. The key constructs to support decentralization of control are coordination mechanism and coordination models. Reflective computations in a decentralized setting coordinate at every stage to share knowledge, analyze the realization of goals, derive plans, and synchronize the execution of adaptations. Coordination mechanisms provide the means to resolve conflicts arisen from dependencies between nodes in a decentralized setting. Coordination models capture the runtime data required for coordination.

Our experiences with the development of decentralized self-adaptive systems, as well as the generalization of the lessons learned in the form of a reference model, have helped us to identify several key challenges in this area. The challenges we have have identified span the interest of numerous research communities, including software engineering, multi-agent systems, artificial intelligence, self-organizing systems, and theory. We thus believe there is a need for an inter-disciplinary approach to challenges posed here. In particular, we believe software engineering community is in a unique position to bridge the gap between these communities—a task that we have begun to undertake in this paper, and more generally pursue in the organization of a workshop series targeted at this problem [34].

Partial Knowledge. The types of analysis and planning performed in this setting needs to deal with the reality that partial knowledge is the norm, rather than the exception. Even in a collaborative setting, sharing complete knowledge between decentralized adaptation managers constrains the scalability of the systems, as outlined in [12]. Moreover, a number of commonly employed quantitative decision making techniques, such as non-linear programming from operations research and queueing network models from performance analysis, are not directly applicable, since they rely on the availability of system-wide knowledge. The lack of complete knowledge forces each self-adaptive unit to make sub-optimal solutions. While through carefully designed algorithms, it may be possible to develop algorithms that converge to (near-)optimal solutions, in practice, engineering real-world solutions in this manner has shown to be extremely difficult for some of the reasons outlined below.

Uncertainty. Any self-adaptive system situated in a dynamic, especially physical, environment has to deal with uncertainty. This is due to the fact that physical environments are by definition unpredictable, hence at any point in time the environment model of a self-adaptive system may become inconsistent with the actual environment, and a self-adaptive system may have no control over other processes that influence the environment. The situation is exacerbated in decentralized self-adaptive systems, where there is no central authority, and the system's organization itself may fluctuate dynamically as a result of self-adaptive units that join and leave the system. Artificial self-organizing systems have shown to be particularly robust to dynamic operating conditions [31, 25]. Exploiting principles from this field is a promising direction to deal with uncertainty in decentralized self-adaptive systems.

Conflicting Goals. In decentralized self-adaptive systems, each self-adaptive unit may pursue its own goals. These systems often share logical and physical resources, and thus their goals may conflict with one another. In such setting, either the self-adaptive units cooperate with one another to achieve a fair solution, or compete with one another to maximize personal gain. Numerous engineering questions in either case arise: How to model, relate, and incorporate individual goals to reason about a system-wide solution? Which conflict resolution mechanism is more suitable for a particular type of quality concern? How to trust individual self-adaptive units to play fairly and not selfishly? The body of work on market mechanisms and negotiation [15] from the field of multi-agent systems is a promising starting point to tackle the complex problems of conflicting goals. Reinforcement learning techniques [16] could also be useful for developing self-adaptive units that over time learn the best approach to adapt and negotiate with one another. In fact, learning may also be employed to address other issues, such as uncertainty (e.g., through learning the emerging properties) and partial knowledge (e.g., through inducing more knowledge based on the available information).

Overhead. As evident in our reference model, the centerpiece of decentralized self-adaptation is coordination. Coordination may occur at different phases and levels. However, coordination in a distributed setting results in additional communication and computation that is of utmost significance from a software engineering perspective. For instance, to deal with partial knowledge, self-adaptive units may choose to coordinate through sharing of local information with one another, while to deal with conflicting goals, self-adaptive units may choose to coordinate through negotiation or voting protocols. Clearly, in both cases one has to consider two issues: (1) the computing inefficiency as a result of resources utilized for coordination, and (2) the timeliness impact of making and effecting adaptation decisions due to such coordination efforts.

Guarantees. Providing system-wide assurances in both self-adaptive and decentralized software is challenging. The situation is exacerbated when one considers self-adaptive software that is also decentralized. The difficulty of guaranteeing the system's behavior is a byproduct of some of the challenges mentioned earlier, e.g., lack of knowledge, and uncertainty. On top of providing assurances for the system's behavior (e.g., timeliness), in some cases ensuring that even a solution can be eventually converged on is challenging. For instance, in many market-based algorithms, if a sensible mechanism design is not employed, no particular solution converges. Moreover, another related issue of concern is unwanted (unaccounted for) emergent behavior. For instance, consider that applying the deployment improvement framework to a highly unstable system may result in continuous redeployment of the system, which may degrade the system's QoS instead of improving it, and the lack of a central authority makes it extremely difficult to detect such situations. Assurances could take on various forms: Is the system guaranteed to achieve a particular set of objectives? Is it guaranteed for the system to achieve those objectives within a certain time limit? Inspiring starting points to deal with guarantees in decentralized self-adaptive systems are formal verification of negotiation protocols [32], Law-Governed Interactions [27], and controlling emergent behavior [30].

Systematic Engineering. Finally, in our experiences we have observed significant trade-offs among the design decisions that address the challenges listed above. For instance, selecting the right mechanism for coordination often depends on the level of overhead that can be tolerated, while the level of overhead depends on the level of guarantees desired. A systematic approach to engineering such systems calls for a better understanding of such trade-offs. To that end, a number of challenging questions need to be addressed: How could we apply the principles of software architecture to the organization of such systems? How the different ways of organizing such systems (e.g., peer-to-peer vs. hierarchical) relate to architectural styles? Is it possible to develop a catalog of organization patterns that promote specific properties? What is the impact of organization patterns on coordination mechanisms utilized in lower levels? We believe the following studies [13, 14, 18, 28] are steps in the right direction.

Acknowledgements

This work is partially supported by grant CCF-0820060 from the National Science Foundation. Danny Weyns is funded by the Research Foundation Flanders (FWO).

7. REFERENCES

[1] J. Andersson et al. Reflecting on self-adaptive software systems. In *Workshop on Software Engineering for Adaptive and Self-Managing Systems*, Vancouver, BC, May 2009.

[2] J. Andesson et al. Modeling dimensions of self-adaptive software systems. In Betty H. C. Cheng et al., editors, *LNCS Hot Topics on Software Engineering for Self-Adaptive Systems*. Springer, 2009.

[3] Y. Brun et al. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems*, volume 5525, pages 48–70. Lecture Notes in Computer Science Hot Topics, 2009.

[4] B. Cheng et al. Software engineering for self-adaptive systems: A research road map. In Betty H. C. Cheng et al., editors, *LNCS Hot Topics Software Engineering for Self-Adaptive Systems*. Springer, 2009.

[5] S. Cheng et al. Evaluating the effectiveness of the rainbow self-adaptive system. In *Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMSŠ09), Vancouver, BC, Canada*, 18-19 May 2009.

[6] G. Coulson et al. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1):1–42, 2008.

[7] K. Crowston. A taxonomy of organizational dependencies and coordination mechanisms. Working paper series, MIT Center for Coordination Science, 1994.

[8] S. Dobson et al. A survey of autonomic communications. *TAAS*, 1(2):223–259, 2006.

[9] J. Dowling and V. Cahill. The k-Component architecture meta-model for self-adaptive software. In *Int'l Conf. on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 81–88, London, UK, 2001. Springer-Verlag.

[10] G. Edwards et al. Architecture-driven self-adaptation and self-management in robotics systems. In *Workshop on Software Engineering for Adaptive and Self-Managing Systems*, Vancouver, BC, May 2009.

[11] D. Garlan et al. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):276–277, October 2004.

[12] I. Georgiadis, J. Magee, and J. Kramer. Self-Organising Software Architectures for Distributed Systems. In *1st Workshop on Self-Healing Systems*, New York, 2002. ACM.

[13] C.S. Hayden et al. A catalog of agent coordination patterns. In *Annual Conf. on Autonomous Agents*, pages 412–413, New York, NY, USA, 1999. ACM.

[14] B. Horling and V. Lesser. A Survey of Multi-Agent Organizational Paradigms. *The Knowledge Engineering Review*, 19(4):281–316, 2005.

[15] N.R. Jennings et al. Automated Negotiation: Prospects, Methods and Challenges. *Group Decision and Negotiation*, 10(2):199–215, 2001.

[16] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[17] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[18] M. Kolp et al. Multi-agent architectures as organizational structures. *Autonomous Agents and Multi-Agent Systems*, 13(1):3–25, 2006.

[19] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Int'l Conf. on Software Engineering*, May 2007.

[20] P. Maes. Concepts and experiments in computational reflection. In *OOPSLA*, Orlando, FL, Oct 1987.

[21] S. Malek et al. A decentralized redeployment algorithm for improving the availability of distributed systems. In *3rd Int'l Conf. on Component Deployment*, Grenoble, France, 2005.

[22] S. Malek et al. A framework for ensuring and improving dependability in highly distributed systems. *Architecting Dependable Systems III, LNCS*, October 2005.

[23] S. Malek et al. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Transactions on Software Engineering*, 31(3):256–272, 2005.

[24] T.W. Malone and K. Crowston. Toward an interdisciplinary theory of coordination. *ACM Computing Surveys*, 26(1):87–119, 1994.

[25] M. Mamei and F. Zambonelli. *Field-based coordination for pervasive multiagent systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[26] M. Mikic-Rakic et al. A tailorable environment for assessing the quality of deployment architectures in highly distributed settings. *LNCS*, pages 1–17, 2004.

[27] N. Minsky. On conditions for self-healing in distributed software systems. *Autonomic Computing Workshop*, 2003.

[28] A. Oluyomi et al. A comprehensive view of agent-oriented patterns. *Autonomous Agents and Multi-Agent Systems*, 15(3):337–377, 2007.

[29] P. Oreizy et al. Architecture-based runtime software evolution. In *Int'l Conf. on Software engineering*, Kyoto, Japan, May 1998.

[30] H. V. D. Parunak and S. Brueckner. Engineering swarming systems). In *Methodologies and Software Engineering for Agent Systems*. Springer, 2004.

[31] H. V. D. Parunak and S. Brueckner. Analyzing Stigmergic Learning for Self-Organizing Mobile Ad-Hoc Networks (MANET's). In *Engineering Self-Organising Systems, Methodologies and Applications*, Lecture Notes in Computer Science, Vol. 3464. Springer, 2005.

[32] T. W. Sandholm. An implementation of the contract net protocol based on marginal cost calculations. In *12th International Workshop on Distributed Artificial Intelligence*, pages 295–308, Hidden Valley, Pennsylvania, 1993.

[33] M. Shaw. Beyond objects: A software design paradigm based on process control. *ACM SIGSOFT Software Engineering Notes*, 20(1):27–38, January 1995.

[34] SOAR: Workshop Series on Self-Organizing Architectures. `http://distrinet.cs.kuleuven.be/events/soar/2010/`.

[35] F. Tisato et al. Architectural reflection: Realising software architectures via reflective activities. In *Second International Workshop on Engineering Distributed Objects*. Springer, 2001.

[36] D. Weyns et al. The MACODO middleware for context-driven dynamic agent organizations. *TAAS*, 5(1):3.1–3.29, 2010.