On Interacting Control Loops in Self-Adaptive Systems

Pieter Vromant and 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

Control loops in self-adaptive systems are typically conceived as a sequence of four computations: Monitor-Analyze-Plan-Execute (MAPE). During the development of a traffic monitoring system with support for self-healing, we have noticed that simple MAPE loops are not sufficient to deal with the more complex failure scenarios. To manage the adaptations in these scenarios, we extend MAPE loops with support for two types of coordination. First, we introduce support for intra-loop coordination enabling MAPE computations within one loop to coordinate with one another. Intra-loop coordination allows the execution of multiple sub-loops within one control loop. Second, we introduce support for inter-loop coordination enabling MAPE computations across multiple loops to coordinate with one another. Inter-loop coordination allows the MAPE computations of different loops to coordinate the various phases of adaptations. We show how we used the extensions to support self-healing in the traffic monitoring system. We discuss an implementation framework that supports coordination of MAPE loops, and from our experiences offer recommendations for 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, control loop, MAPE, coordination

1. INTRODUCTION

Self-adaptability has been proposed as an effective approach to tackle the increasing complexity of managing modern-day software systems. Self-adaptation endows a software system with the capability to deal autonomously with internal dynamics as well as dynamics in the environment [2]. Self-adaptation is realized by adding adaptation logic to a managed system. The adaptation logic typically found in self-adaptive systems comprises a sequence of four computations: monitor, analyze, plan, and execute.

A monitor computation gathers information from the managed system and possibly the system's environment in order to update a set of relevant models, providing the subsequent computations of the control loop with the necessary data. An analyze computation examines the data previously gathered by the monitor computation, and based on the adaptation goals draws conclusions on which further actions should be undertaken by the self-adaptive system. A plan computation puts together a series of adaptation actions to resolve the problem identified by an analyze computation. This set of actions to the managed system is then carried out by an execution computation. A simple sequence of these four reflective computations–Monitor-Analyze-Plan-Execute (MAPE)–is the most obvious way of forming a control loop in a self-adaptive system.

In this position paper, we argue that simple MAPE loops are not always a sufficient solution to deal with more complex adaptations. To manage the adaptations for self-healing in a traffic monitoring system, we extend MAPE loops with support for two types of coordination: one for intra-loop coordination and one for inter-loop coordination. Intra-loop coordination enables MAPE computations within one loop to coordinate with one another, allowing multiple sub-loops within one control loop. For example, to support adaptation scenarios in which the complete planning and execution of the adaptation is not known upfront, we use multiple plan-execute sub-loops after the initial monitoring and analysis phase. Inter-loop coordination enables MAPE computations across loops to coordinate the various phases of adaptations. For example, to support adaptations in which multiple nodes are involved, the planning computations of the MAPE loops at the different nodes may have to coordinate to prepare the execution of adaptation.

The remainder of this paper is structured as follows. In section 2, we introduce the traffic monitoring system and explain a number of failure scenarios. Section 3 motivates the two MAPE loop extensions, and explains in detail how the extensions enable us to deal with a concrete failure scenario in the traffic monitoring system. In section 4, we give an overview of an implementation framework for interacting control loops, and explain how we have applied it to the traffic monitoring system. We discuss related work in section 5 and discuss challenges ahead in section 6.

2. TRAFFIC MONITORING SYSTEM

The traffic monitoring system consists of a set of intelligent cameras which are distributed evenly along the road. An example of a highway is shown in Fig. 1. Each camera has a limited viewing range and cameras are placed to get an optimal coverage of the highway with a minimum overlap. The task of the cameras is to

SEAMS '11, May 23-24, 2011, Waikiki, Honolulu, HI, USA

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, etc. Our particular focus here is on self-healing of silent node failures, i.e., failures in which a failing camera becomes unresponsive without sending any incorrect data. Since there are dependencies between the software running on different cameras (we explain this in detail below), such failures may bring the system to an inconsistent state and disrupt its services. In [9], we studied a simple failure scenario that can be resolved by local adaptations. In this paper, we consider more complex types of failure events that require coordination between cameras to heal the system. The aim of self-healing is to bring the system to a consistent state and restore its services, although in degraded mode since the traffic state is no longer monitored in the viewing range of a failing camera.

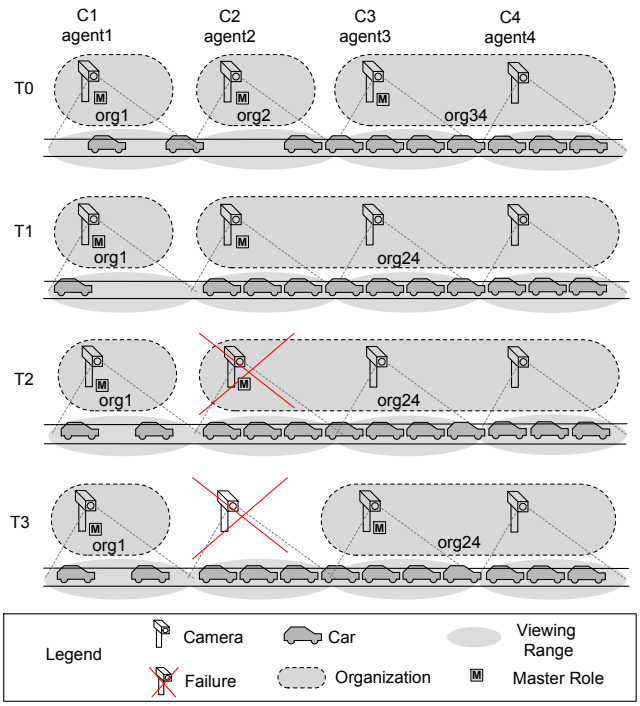

Figure 1: Self-healing scenario

2.1 Domain Functionality

Figure 2 shows the primary components of the software deployed on each camera, i.e. the *local camera system*. The *local traffic monitoring system* provides the domain functionality to detect traffic jams and inform clients. The local traffic monitoring system is conceived as an agent-based system consisting of two components. The *agent* is responsible for monitoring the traffic and collaborating with other agents to report a possible traffic jam to clients. In normal traffic conditions, each agent belongs to a single member *organization*. However, when a traffic jam is detected that spans the viewing range of multiple neighboring cameras, organizations on these cameras will merge into one organization. To simplify the management of organizations and interactions with clients, the organizations have a master/slave structure. The master is responsible for managing the dynamics of that organization (merging and splitting) by synchronizing with its slaves and with the masters of neighboring organizations. Therefore, the master uses the context information provided by its slaves about their local monitored traffic conditions. At T0, the example in Fig. 1 shows two single member organizations, *org1* with *agent1* and *org2* with *agent2*, and one merged organization, *org34* with *agent3* and *agent4*. At T1, the traffic jam spans the viewing range of cameras 2, 3 and 4. As a result, organizations *org2* and *org34* have merged to form *org24*. When the traffic jam resolves, the organization is split dynamically. The *organization middleware* offers services for agents to set up and maintain organizations. To access the hardware and communication facilities on the camera, the local traffic monitoring system can rely on the services provided by the *distributed communication and host infrastructure*.

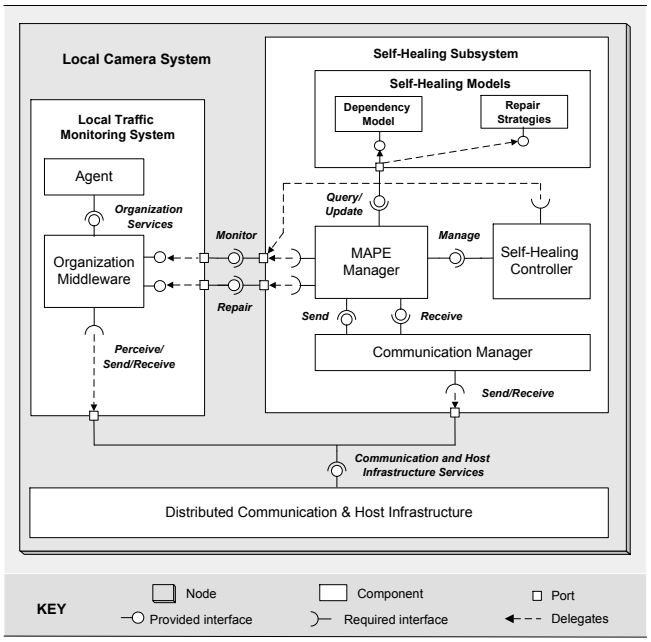

Figure 2: Primary components of a local traffic camera

2.2 Self-Healing Scenarios

Looking at the traffic monitoring system, we can see that each camera occupies one of three distinct roles at any given time: master of a single member organization, master of an organization with additional slaves, or slave in an organization. As these roles come with certain responsibilities, each camera is dependent on a particular set of remote cameras in order to function properly.

1. The master of a single agent organization is dependent on its neighboring nodes and the masters of neighboring organizations.
2. The master of an organization with multiple cameras is dependent on its neighboring nodes, the slaves of its organization and the masters of neighboring organizations.
3. The slave of an organization is dependent on its neighboring nodes and the master of its organization.

Consequently, every failure of a camera is of concern to a number of other cameras, in particular, the cameras with a dependency to the role of the failing camera. Specifically, the traffic monitoring system can experience the following failure events:

1. The master of a single agent organization fails. This event affects the neighboring nodes and the masters of neighboring organizations.
2. The master of an organization with multiple cameras fails. This event affects the neighboring nodes, the remaining slaves of the organization and the masters of neighboring organizations.
3. The slave of an organization fails. This event affects the neighboring nodes and the master of the organization.

We define a self-healing scenario as "an integrated strategy on how to deal with a particular node failure event." The role of the failing camera determines the type of failure and as such the set of cameras that are involved in the failure. These cameras will take part in the self-healing scenario designed to deal with that event. Note that based on the role a camera is occupying and the dependencies that this role implies, a camera may be involved in multiple self-healing scenarios. For example, a slave is involved in a scenario to deal with a failure of the master of its organization, but also in scenarios to deal with failures of its neighboring cameras.

As an example, Fig. 1 shows a failure of camera 2 at T2. This corresponds to failure event 2, since camera 2 serves as the master of an organization with multiple cameras. The self-healing managers with dependencies to camera 2, i.e. the slaves of organization *org24*, i.e. cameras 3 and 4, the masters of the neighboring organizations, i.e. camera 1, and the neighboring cameras 1 and 3, will detect the failure and repair it as defined in the respective self-healing scenarios. At T3 the system has recovered from the failure and can continue its correct operation. Camera 3 is the new master of organization *org24* and the masters of *org1* and *org24* have adapted their respective dependencies with the masters of neighboring organizations. In addition, cameras 1 and 3 have also adapted their neighbors in the dependency model. We discuss a concrete self-healing scenario for this failure in detail in section 3.

2.3 Self-Healing Subsystem

To recover from camera failures, a *self-healing subsystem* is added to the local traffic monitoring system, as shown in Fig. 2. A self-healing subsystem comprises the following components:

- *Self-Healing Models* which consist of two models that can be inspected and updated via the *Query/Update* interface:
  1. *Dependency Model* contains a model of the current dependencies of the components of the local traffic monitoring system with other active cameras. Dependencies include master/slave relationships, neighbor relationships, etc.
  2. *Repair Strategies* contains, for different types of failure scenarios, the repair actions required to bring the traffic monitoring system back to a consistent state. Examples of repair actions are: halt the communication of the local traffic monitoring system with the failing camera, remove a slave of the failing camera from the list of slaves, elect a new master, exchange the context with another camera, change the neighbor to the camera next to the failing camera, etc.
- *Self-Healing Controller* ensures that the self-healing subsystem deals with the correct failure scenarios corresponding to the actual role of the local traffic monitoring system. The self-healing controller observes the role of the local traffic monitoring system using the *Monitor* interface. As discussed, we distinguish between three different roles: master of a single member organization, master with slaves, and slave. Via the *Manage* interface, the self-healing controller instructs the MAPE manager to deal with the set of failure events that affect the local traffic monitoring system in the current role. Therefore, the MAPE manager employs the necessary repair strategies to participate in the proper set of self-healing scenarios.
- *MAPE Manager* contains the logic to deal with self-healing. Depending on the actual role of the local monitoring system, different types of failure events will be observed and dealt with based on the corresponding repair strategies. The MAPE manager monitors the main system using the *Monitor* interface, thereby maintaining the dependency model. To detect failures, the MAPE manager coordinates with self-healing subsystems on other cameras with a dependency in the dependency model using the *Send* and *Receive* interfaces. When a failure is detected, the manager may need additional coordination with other managers before it repairs the local traffic monitoring system. For example, when a slave detects that the master of its organization fails, it initiates an election protocol with the other slaves of the organization to elect a new master. Once a new master is established, the necessary repair actions are executed via the *Repair* interface, bringing the local traffic monitoring system back in a consistent state. We discuss the detailed design of the MAPE manager in section 4.
- *Communication Manager* facilitates intra-loop and inter-loop communication between computations of MAPE managers. The *Send* interface offers facilities for computations of the MAPE manager to send coordination messages. Messages can be exchanged either with local MAPE computations (intra-loop coordination) or with remote MAPE computations (inter-loop coordination). Local messages are directly delivered via the *Receive* interface. Remote messages are forwarded to the distributed communication infrastructure via the *Send/Receive* interface. Remote messages that arrive at the communication manager are passed to the local *MAPE manager* via the *Receive* interface.

3. MAPE LOOP EXTENSIONS

In this section, we take a closer look at one particular failure event in the traffic monitoring system and at the self-healing scenario designed to deal with that event. The scenario is defined in terms of interacting Monitor-Analyze-Plan-Execute (MAPE) control loops, and illustrates how the proposed coordination extensions to the typical MAPE loop are integral to the process of bringing the main traffic monitoring system back to a consistent state and allowing it to function properly again. Intra-loop coordination supports multiple sub-loops within a single control loop, and inter-loop coordination supports the coordination of adaptations across control loops.

We focus on failure event 2, meaning the failure of a master of an organization with multiple cameras. The diagram in Fig. 3 illustrates how control loops running on different cameras interact in this particular scenario to perform a series of adaptations bringing the traffic monitoring system back to a consistent state after the failure of a master of an organization. The first number of the subscript of each MAPE computation is an identifier of the failure scenario. The second number of plan (P) and execute (E) identifies the sub-loop the computations belong to. As shown on top of the diagram, the failing camera is a master with two slaves. Its organization also borders on a different organization, governed by a separate master.

In section 2 we mentioned that in the event of a failure of the master of an organization with multiple cameras, the remaining slaves as well as the masters of all neighboring organizations are affected. In this scenario, control loops are therefore running on these two types of nodes, in addition to the loop on the camera with the subject, i.e. the master for which this particular self-healing scenario has been set up. The scenario is made up of five parts:

1. A monitoring and analysis sub-loop in which the silent failure of a camera which is the master of an organization of traffic cameras is detected. Therefore, the monitor computations of the MAPE managers of all cameras that are dependent on the subject camera of the scenario periodically ascertain its status using the ping/echo protocol. At this stage, the subject camera is still an active part of the scenario. As soon as that node fails, however, the local analyze computations on each dependent camera will detect the absence of ping-replies and initiate the rest of the scenario.

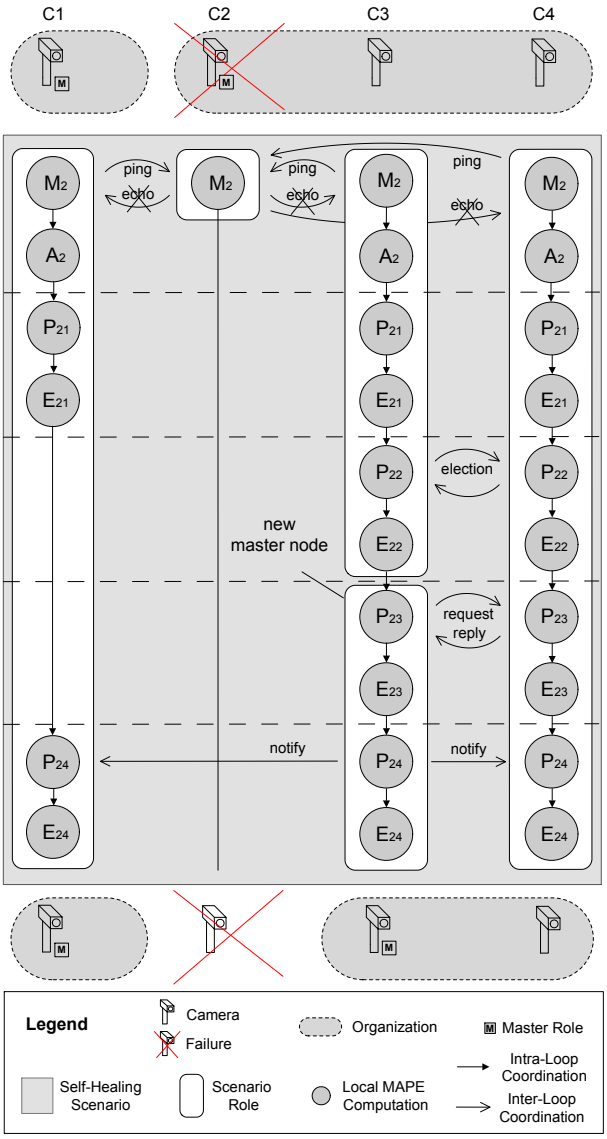

Figure 3: Master failure scenario

2. A first planning and execution sub-loop in which the slaves of the failing master and the master of the neighboring organization instruct their local traffic monitoring system to halt the communication with the local traffic monitoring system of the failing camera which is now known to be offline. This part of the scenario requires no inter-loop coordination between computations of different MAPE loops: all dependent cameras are already aware of the failure of the master and temporarily pause the relevant communications of the local traffic monitoring systems.
3. A second planning and execution sub-loop in which the remaining slave nodes coordinate to elect one slave as the new master of their organization using an appropriate election protocol. Once a new master is elected, the organizational change is effected to the local traffic monitoring system of each involved camera.
4. A third planning and execution sub-loop in which the newly elected master is prepared to take up its new role. The newly elected master collects and processes the local traffic contexts from each of its slaves using a simple request/reply protocol.
5. A fourth planning and execution sub-loop in which the new master notifies both its slaves and the master of the neighboring organization to resume the base-level communication after which the system can resume normal operation.

4. DESIGN & IMPLEMENTATION

Having elaborated on the concept of self-healing scenarios, we now explain a concrete implementation framework for the intra-loop and inter-loop MAPE extensions based on the primary architecture shown in Fig. 2. Subsequently, we explain the self-healing controller and the MAPE manager, and how we used them to implement the traffic monitoring system.

4.1 Self-Healing Controller

Figure 4 shows the internal components of the self-healing controller. *Role Monitor* uses the *Monitor* interface to determine the current role of the local traffic monitoring system. When the role changes the role monitor informs *Role Manager* via the *Role Change* interface. Role manager then instructs the MAPE manager to adapt the strategies for dealing with the failure events relevant to the newly adapted role. The *Manage* interface supports adaption of the various phases of the MAPE loop:

```
void activate(MAPEComponentType c, Role r)
```

MAPE component types are *monitor*, *analyze*, *plan*, and *execute*, and roles are *single master*, *master with slaves*, and *slave*.

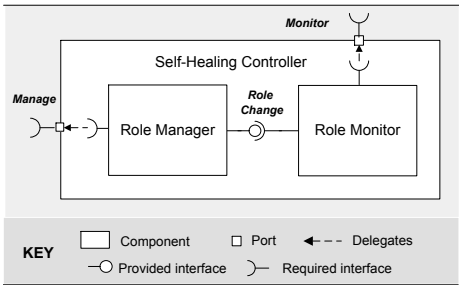

Figure 4: Components of the self-healing controller

4.2 MAPE Manager

Figure 5 shows the components of the MAPE manager. Each MAPE component (*Monitor*, *Analyze*, *Plan*, and *Execute*) encapsulates a corresponding MAPE computation. The various components execute the appropriate parts of the repair strategies for the current role of the local traffic monitoring system. Role changes are notified via the *Manage* interface. During the execution of the strategies, the MAPE components can coordinate with other components using the corresponding *Coordination Manager*.

Figure 6 zooms in on the coordination manager for the monitor component. The coordination manager assists the MAPE component to asynchronously coordinate with other MAPE components in the context of particular self-healing scenarios.

The *Coordination Point Manager* offers facilities for the MAPE component to register new self-healing scenarios with other components via the *Coordinate* interface:

```
void register(Scenario s);
```

Scenario objects are defined as follows:

```
Scenario = < CameraID subject, FailureType ft >
```

A scenario object uniquely identifies a self-healing scenario based on an identifier of the subject of the failure and the failure type. Failure types are *single master failure*, *master with slaves failure*, and *slave failure*.

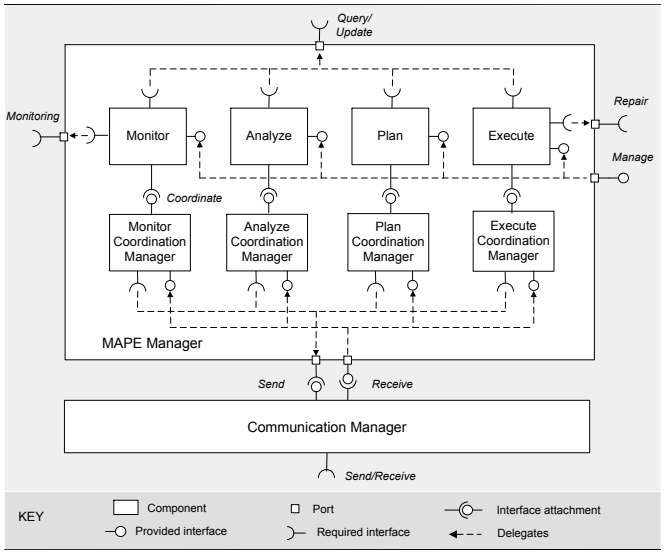

Figure 5: Components of the MAPE manager

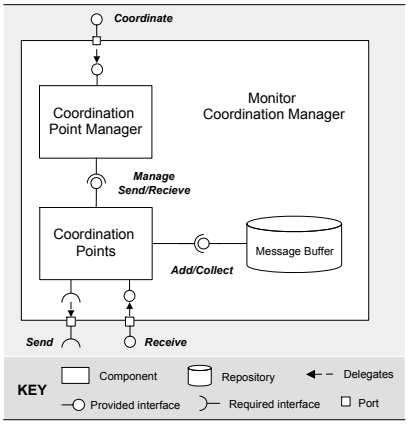

Figure 6: Components of a coordination manager

Registration of the scenario instantiates a new *Coordination Point* and adds it to the set of *Coordination Points*. A coordination point manages the coordination for the MAPE component and a particular scenario. After registration of the scenario, the coordinate interface enables the MAPE component to exchange self-healing coordination information with other MAPE components:

```
void send(SelfHealingMessage m);
SelfHealingMessage receive(Scenario s);
```

A self-healing message is structured as follows:

```
SelfHealingMessage = < SenderNodeID sid,
                       RecipientNodeID rid,
                       MAPEComputationID id,
                       Scenario s,
                       CoordinationData d >
```

MAPE computation ID is a unique identifier of the MAPE computation. Examples are M_2 and P_{23} as shown in Fig. 3. Coordination data consist of the performative specific to the step in the coordination protocol and the actual data that is exchanged. For instance, the intra-loop coordination data sent from P_{22} to E_{22} is `transition(CameraID newlyElectedMaster)`. Examples of inter-loop coordination data for the scenario are `ping()` and `echo()` for M_2, and `reply(TrafficState congested)` for P_{23}.

The coordination point manager dispatches coordination messages via the *Manage Send/Recieve* interface to the proper coordination point. The coordination point can send coordination messages to other MAPE components via the *Send* interface of the *Communication Manager*. The communication manager passes messages for local MAPE components directly to the proper coordination manager via the *Receive* interface. Messages for remote MAPE components are forwarded via the distributed communication infrastructure via the *Send/Receive* interface.

The communication manager dispatches messages (locally received via the send interface or remotely via the send/receive interface) to the proper coordination manager via the *Receive* interface. The coordination point of the corresponding failure scenario adds messages to the *Message Buffer* where they wait for delivery. MAPE components can collect messages per failure scenario by invoking the receive operation of the coordinate interface. The coordination point manager uses the corresponding coordination point to collect a message from the buffer and deliver it to the MAPE component.

Figure 7 shows how the various components interact during a specific part of the master failure scenario, i.e. part 3 of the scenario in Fig. 3. Excerpts of both the election of a new master by the plan components (inter-loop coordination), and the local transition at slave 1 from plan to execute (intra-loop coordination) are shown. During the election, the P_{22} computation of the plan component of slave 3 (i.e. the slave of camera C3, see Fig. 1) sends an election message to the remaining slave 4 of camera C4 that contains the camera identifier. Therefore, P_{22} invokes the send operation of the plan coordination manager. The message is handled by the appropriate coordination point that sends the election message to the local communication manager. The local communication manager forwards the message to the distributed communication infrastructure for delivery at slave 4. Similarly, slave 4 sends an election message to slave 3. When the election message from slave 4 arrives at camera C3, the communication manager delivers it to the proper coordination manager. It does this by verifying the message's MAPEComputationID, in this case P_{22}, and its scenario information, which contains the id for the failing camera and the master node failure scenario type. P_{22} collects the election message by invoking the receive operation for the given failure scenario. As a rule, the slave with the highest camera id will become the new organization master. Therefore, P_{22} compares the identifiers and marks the camera with the highest id as preliminary master. After a predefined time window (electionTimeout) without having received additional messages, the election protocol ends and P_{22} marks the temporal master as the new master of the organization. Subsequently, P_{22} sends an intra-loop transition message, again via plan coordination manager and the communication manager. When computation E_{22} of the execute component receives this message, it executes the repair action, adapting the organization of the local traffic monitoring system to the new organizational structure. Camera C3 becomes the new master with camera C4 as slave.

5. RELATED WORK

Hebig et. al. [4] argue to make control loops explicit and present a UML profile for control loops that extends UML modeling concepts. The profile allows the specification of interactions between control loops at the level of the controller, while the work presented in this paper zooms in on the coordination between specific MAPE computations within and across control loops.

Villegas and Müller [7] present a reference model for context-based self-adaptation that supports exchanging context information between different control loops to support their operations. E.g., context information resulting from the analysis in one loop is pro-

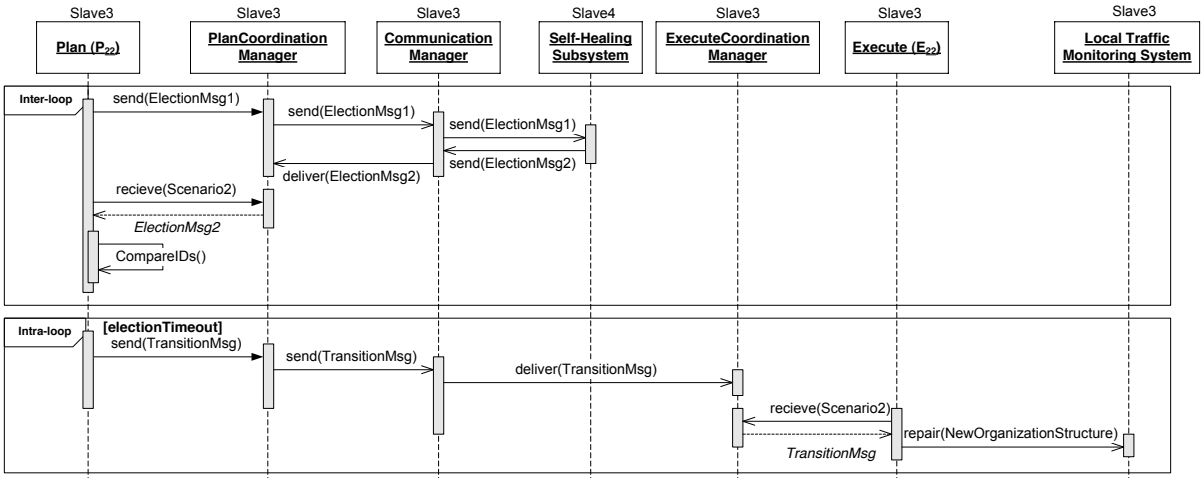

Figure 7: Coordination between MAPE computations when electing a new master (part 3 of Fig. 3)

vided to support the decision making in another loop. Our work goes further in offering support for coordination between the computations of different control loops.

Several other authors have studied particular aspects of coordination of multiple control-loops in self-adaptive systems. A concrete example from task coordination for a self-adaptive system in the robotics domain is discussed by Edwards et al. [3]. Raheja et. al. [6] discuss how to coordinate multiple loops, responsible for multiple concerns (tasks), using preemption in Rainbow. Malek et al. [5] use an auction based coordination mechanism to coordinate resource usage for a self-adaptive distributed system that utilizes adaptive (re)-deployment. Coordination of multiple control-loops responsible for context-aware composition and on-line learning for autonomic software product lines is discussed by Abbas et. al. [1]. In comparison to these works, this paper provides a more focused and comprehensive discussion to the general problem of integrating and coordinating multiple control-loops.

6. DISCUSSION & CHALLENGES AHEAD

Our experience with a self-adaptive software system demonstrated the limitations of a MAPE loop with accurately explaining and representing coordination in a distributed setting. We reconceptualized MAPE by enhancing it with two types of coordination (inter-loop and intra-loop), which resulted in a more natural way of designing and realizing self-adaptation. We described an implementation framework aimed at streamlining the development of both types of coordination in a distributed setting[1].

While our experience in the context of a self-managing traffic monitoring system has been very positive, several avenues of future research remain. In particular, we are exploring the extent in which the framework can be employed in other application domains. To that end, we plan to develop an extensible architecture in which modules realizing different coordination mechanisms (e.g., event driven, method-call driven) and protocols (e.g., voting, auctioning, gossip) are incorporated into the framework, and made available as reusable library for the community. Furthermore, the current framework assumes that only one adaptation can happen at a time. We plan to extend the framework architecture with support for concurrent adaptations beyond a single concern (e.g. self-healing and self-optimization in parallel). In the traffic monitoring system, each control loop comprises the four MAPE computations. Obviously, other configurations are possible, such as a setting where monitoring and execution are distributed, but analysis and planning are centralized. We plan to study how the MAPE manager can be extended to support different patterns of distribution of MAPE computations.

Finally, we also aim to develop a more rigorous approach for specifying various coordination patterns of MAPE loops that are possible in self-adaptive systems. Therefore, we plan to integrate the coordination patterns with our earlier work on a formal language for specifying self-adaptive systems [8]. We believe the resulting language would allow one to formally specify both structural and behavioral properties of a self-adaptive system, providing a basis for expressing architectural choices and evaluate alternative designs.

[1] http://people.cs.kuleuven.be/danny.weyns/MAPEcode.zip.

7. REFERENCES

[1] N. Abbas et al. Autonomic software product lines. In *International Workshop on Variability in Software Product Line Architectures*, Copenhagen, 2010.

[2] B. Cheng et al. Software engineering for self-adaptive systems: A research road map. In *Software Engineering for Self-Adaptive Systems, LNCS vol.5525*, 2009.

[3] G. Edwards et al. Architecture-driven self-adaptation and self-management in robotics systems. In *Software Engineering for Adaptive and Self-Managing Systems, SEAMS*, 2009.

[4] R. Hebig, H. Giese, and B. Becker. Making control loops explicit when architecting self-adaptive systems. In *Self-Organizing Architectures, LNCS vol. 6090*, 2010.

[5] S. Malek et al. A decentralized redeployment algorithm for improving the availability of distributed systems. In *Int. Conference on Component Deployment*, France, 2005.

[6] R. Raheja et al. Improving architecture-based self-adaptation using preemption. In *Self-Organizing Architectures, LNCS vol. 6090*, 2010.

[7] N. Villegas and H. Müller. A control engineered reference model for context-based self-adaptation. In *Dagstuhl Seminar 10431 on Self-adaptive Systems*, 2010.

[8] D. Weyns, S. Malek, and J. Andersson. FORMS: a formal reference model for self-adaptation. In *7th Int. Conf. on Autonomic Computing ICAC*, Washington, DC, 2010.

[9] D. Weyns, S. Malek, and J. Andersson. On decentralized self-adaptation: Lessons from the trenches and challenges for the future. In *Software Engineering for Adaptive and Self-Managing Systems, SEAMS*, Cape Town, 2010.