FORMS: Unifying Reference Model for Formal Specification of Distributed Self-Adaptive Systems

Danny Weyns, Linnaeus University, Sweden
Sam Malek, George Mason University, USA
Jesper Andersson, Linnaeus University, Sweden

The challenges of pervasive and mobile computing environments, which are highly dynamic and unpredictable, have motivated the development of self-adaptive software systems. Although noteworthy successes have been achieved on many fronts, the construction of such systems remains significantly more challenging than traditional systems. We argue this is partially because researchers and practitioners have been struggling with the lack of a precise vocabulary for describing and reasoning about the key architectural characteristics of self-adaptive systems. Further exacerbating the situation is the fact that existing frameworks and guidelines do not provide an encompassing perspective of the different types of concerns in this setting. In this paper, we present a comprehensive reference model, entitled FOrmal Reference Model for Self-adaptation (FORMS), that targets both issues. FORMS provides rigor in the manner such systems can be described and reasoned about. It consists of a small number of formally specified modeling elements that correspond to the key concerns in the design of self-adaptive software systems, and a set of relationships that guide their composition. We demonstrate FORMS's ability to precisely describe and reason about the architectural characteristics of distributed self-adaptive software systems through its application to several existing systems. FORMS's expressive power gives it a potential for documenting reusable architectural solutions (e.g., architectural patterns) to commonly encountered problems in this area.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures

General Terms: Design, theory

Additional Key Words and Phrases: Formal Methods, Self-Adaptation, Autonomic Computing

1. INTRODUCTION

Pervasive, mobile, and embedded computing environments are characterized by a high degree of unpredictability and dynamism in the execution context. These environments call for a new class of software systems, known as self-adaptive software system. Self-adaptability endows a software system with the capability to adapt its behavior at runtime to changes in its execution conditions and user requirements [Kephart and Chess 2003; Kramer and Magee 2007].

The development of self-adaptive software systems has shown to be significantly more challenging than traditional systems [Cheng et al. 2009]. Over the past decade numerous solutions for alleviating the situation have been developed. In particular, researchers and practitioner have proposed several frameworks for constructing such systems. Some have aimed to serve as conceptual guidelines, such as IBM's MAPE-K [Kephart and Chess 2003] (Monitor-Analyze-Plan-Execute-Knowledge) that describes the different stages of self-adaptation, the work of Mary Shaw [Shaw 1995] that recognizes the feedback-control loop as an essential process within any self-adaptive system, and the architectural model of Kramer and Magee [Kramer and Magee 2007] that puts component control, change management, and goal management in three distinct layers. Others have adopted an implementation perspective, such as Archstudio [Oreizy et al. 1998], Rainbow [Garlan et al. 2004], and MUSIC [Geihs et al. 2009], which advocate a software architecture-based approach for assessing the adaptation decisions and making the changes.

All of the above models and frameworks have been intended to serve merely as guidelines, and provide significant leeway in how the engineer architects the software system. For instance, given any one of these frameworks, the same functionality may be realized using starkly different architectures (e.g., centralized vs. decentralized, flat vs. hierarchical). Therefore, while these frameworks have achieved noteworthy success in many domains, they are neither formal enough to unambiguously describe and reason about the primary architectural characteristics of self-adaptive systems, nor is that their intended use. At the same time, each framework has targeted a particular set of concerns, which we informally refer to as *perspective*. None of the frameworks provides a rich enough set of elements for describing the different types of perspectives.

The hallmark of any established engineering field is the ability to precisely express and reason about the architectural choices—a capability that is currently lacking in the domain of self-adaptive software, as argued by us [Andersson et al. 2009b] as well as many others [Cheng et al. 2009]. We have begun to address this issue in our previous work [Andersson et al. 2009a; Weyns et al. 2010a] which led to the development of a preliminary reference model for self-adaptation aimed at bringing the differences among such systems to the forefront of the design. However, the reference model proposed in our earlier work [Weyns et al. 2010a] has several limitations; most notably, it is not sufficiently expressive for describing the variations among a large class of self-adaptive software systems that are distributed. This is an issue that has been overlooked not only in our initial reference model, but also by other commonly employed frameworks and guidelines [Weyns et al. 2010a]. One of the key contributions of this paper is the extension of our initial reference model with additional constructs and relationships necessary for describing distributed self-adaptive systems. For the first time, we also provide a comprehensive and detailed description of the reference model, including a full formal specification. Finally, we demonstrate its ability to precisely describe and reason about the primary architectural characteristics of several self-adaptive systems developed in our respective research groups.

The reference model, entitled FORMS, short for FOrmal Reference Model for Self-adaptation, enables software engineers to rigorously describe and reason about the architectural characteristics of distributed self-adaptive systems. FORMS builds on existing frameworks and established principles of self-adaptation, such as computational reflection [Maes 1987], MAPE-K [Kephart and Chess 2003], and architecture-based adaptation [Oreizy et al. 1998; Kramer and Magee 2007]. The reference model offers a vocabulary that consists of a small number of primitives and a set of relationships among them that delineates the rules of composition. The model is formally specified, which enables the engineers to precisely define the key characteristics of self-adaptive software systems, and reason about them.

Through applying FORMS to several existing systems we have confirmed its ability to illuminate the key characteristics of these systems. However, we do not argue FORMS is a conclusive reference model. In fact, one of the key contributions of FORMS is its ability to accommodate future extensions. To ensure extensibility, as well as technology and implementation independence, the primitives are intentionally high-level (i.e., remain at the architectural level) and could be specialized for specific application domains. The primitives refined in this manner enable the engineers to derive and document a catalog of known solutions (e.g., in the form of architectural patterns) for different domains.

While FORMS offers a formally founded vocabulary for the key architectural constructs comprising self-adaptive systems, it does not provide an implementation framework from which self-adaptive applications can be derived. FORMS supports engineers with describing the key concerns of their architectures and reason about important properties, via supporting tools. FORMS can be useful in various scenarios, such as to understand the key architectural decisions of a self-adaptive system in early design or in preparation for a system evolution, to document such decisions for developers, to specialize FORMS for describing and reasoning about specific concerns of self-adaptive systems in particular domains, to employ FORMS as a unifying vocabulary to study self-adaptive systems, etc.

The remainder of this paper is organized as follows. Section 2 presents an example to illustrate the issues and describe the FORMS concepts. Section 3 presents the integration of three perspectives that form the basis of FORMS and describes the corresponding reference models. Section 4 presents our experiences with using the FORMS reference model in a case study. The paper concludes with an overview of related work, a discussion on applications and contributions of FORMS, and future avenues of research in Sections 5, 6, and 7, respectively. The complete formal specification of FORMS and its applications to two additional case studies is provided in the Appendix.

2. ILLUSTRATIVE EXAMPLE

We consider a system from the robotics domain as our illustrative example. The illustrative system is motivated by [Edwards et al. 2009]. The authors propose a layered approach for the design and implementation of self-adaptive behavior of a robotic system. The self-adaptive behavior in this application ensures that the system itself resolves failures of the control software of the robots. This is a representative example for a small scale, distributed, self-adaptive system, i.e., it will change its structure and behavior at run-time in response to changes in the environment or the system itself.

In particular, the adaptive robotic software architecture consists of (1) a basic bottommost layer with the application components that control the robot, and (2) one or more meta-layers with adaptation logic that implement fault-tolerance, dynamic software updates (component replacement), resource discovery, re-deployment, etc. In the proposed architecture, each layer may adapt the layer beneath. The *robot behavior* (bottommost layer) provides the robot's application logic. In a common instance the system is distributed on two or more robots (nodes), where follower robots trail a leader robot. On top of that, using meta-level components, there is a distributed *failure manager* layer that, based on the collected data, detects and resolves failures in the application subsystem. The failure manager layer is the subject to a *version manager* layer, which replaces the *failure collector* components on *robot follower* nodes whenever new versions are available.

In this system, self-adaptation is a key factor for successful deployment of the system, which requires the ability to precisely describe the system architecture, reason about the key design decisions. The challenge is in providing a vocabulary that is sufficiently expressive and precise, while still accessible by and useful for developers. The current practice of expressing the architectural design of self-adaption, and even more importantly documenting the known solutions (e.g., architectural solutions to common problems), is often ad hoc. This is precisely the motivation for our work. We continue this discussion, exemplified with concrete excerpts of the illustrative example, as we present the details of FORMS next.

3. UNIFYING FORMAL REFERENCE MODEL

The work presented in this paper is based on the experience with constructing self-adaptive systems in our research groups and a careful study of the existing literature, including [Kephart and Chess 2003; Kramer and Magee 2007; Oreizy et al. 1998; Garlan et al. 2004; Edwards et al. 2009; Dowling and Cahill 2001; Geihs et al. 2009]). In [Andersson et al. 2009a] we provided a classification of self-adaptive software systems, which helped us with identifying the prominent concerns in self-adaptation. Our study indicates that each of the existing commonly employed frameworks targets a particular set of concerns, referred to as *perspective* in this paper, while ignoring some others. An exhaustive reference model covering all of the different perspectives found in the literature is beyond the scope of this paper, and perhaps infeasible to achieve. Instead our intention has been to establish a reference model covering sufficiently wide spectrum of perspectives, while remaining extensible for future refinements. To that end, we found five key requirements for the specification of self-adaptation capabilities in a given system. In particular, the reference model should have the ability to describe and reason about:

(1) How the system monitors the environment (i.e., context-awareness);
(2) How the system monitors itself (i.e., self-awareness);
(3) How the system adapts itself;
(4) How the system coordinates monitoring and adaptation in a distributed setting.
(5) In addition, the model should have the ability to extend and refine the FORMS primitives for additional concerns and domain-specific concepts.

These requirements along with our previous survey of the field, helped us to identify three commonly employed adaptation perspectives as the basis for FORMS: reflective computation [Maes 1987; Andersson et al. 2009b], distributed coordination [Malone and Crowston 1994; Wooldridge and Jennings 1995; Ossowski and Menezes 2006], and MAPE-K [Kephart and Chess 2003].

While these three perspectives are representatives of radically different concerns, we do not argue that they are the only plausible ones. However, we have strived to provide as comprehensive reference model as possible by unifying the three aforementioned perspectives. We believe a similar approach could be applied to further enrich FORMS with additional, potentially domain specific, concerns.

For readability purposes, we describe FORMS using semi-formal UML diagrams in the paper. Though intuitive, the visual representation does not give a precise semantic description of the constructs, which is exactly why a formal representation of FORMS in Z notation is provided in the Appendix. Z is a standardized formal specification language (ISO/IEC 13568:2002) that builds on set theory and first order predicate logic to precisely specify the primitives without delving into the implementation details. The formal specification is type checked using Community Z Tools [CZT 2010]. We use excerpts of the Z specification to illustrate how the model supports reasoning about a self-healing property of the example in section 4.

3.1 Reflection Perspective

Computational reflection is an established and well-understood concept in programming-in-the-small [Maes 1987]. It has traditionally been studied at the level of programming languages and realized using compiler technologies. However, the principles of computational reflection are also applicable to programming-in-the-large, which represents the complex self-adaptive software systems we are interested in our research [Andersson et al. 2009b]. In fact, numerous previously developed self-adaptation approaches (e.g., [Cazzola et al. 1999; Tisato et al. 2001; Blair et al. 2004]) are based on the principles of reflection.

Figure 1 provides an overview of the FORMS's primitives and their relationships to the reflection perspective. A Z specification of the perspective is provided in Appendix A. As shown in Figure 1, a *self-adaptive system* is situated in an *environment*, and comprises one or more *base-level* and *reflective subsystems*. The environment consists of *attributes* and *processes*. An attribute is a perceivable characteristic of the environment. A process is an activity that can change the environment attributes. For instance, attributes for a robot may correspond to the location of an obstacle, while the movement of a robot is a process that changes the location of that robot. The environment may correspond to both physical and logical entities. Therefore, the environment of a computing system may itself be another computing system. For example,

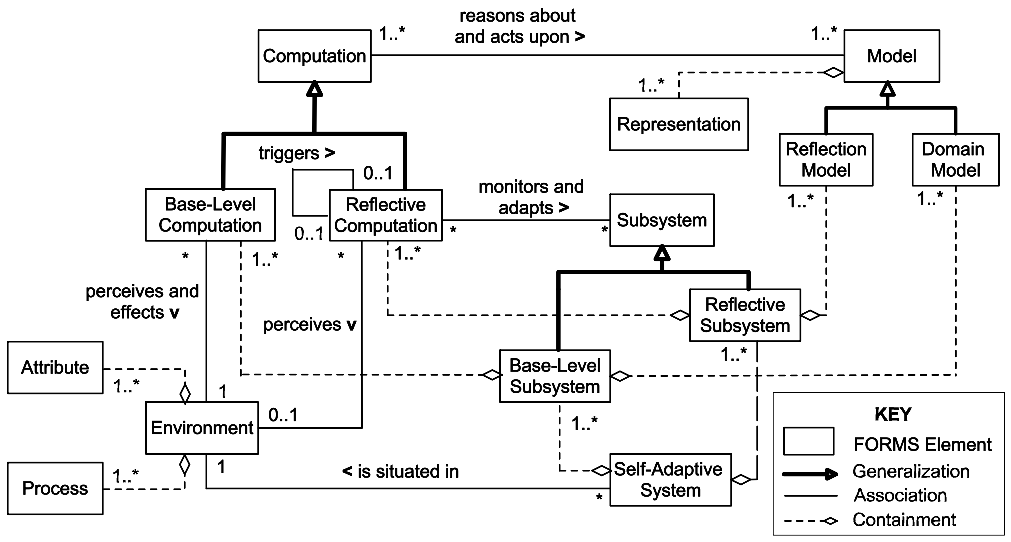


Fig. 1. FORMS primitives that are derived from the computational reflection perspective.

the environment of a robot includes the physical entities like obstacles on its path and other robots, as well as an external mountable camera and the corresponding software drivers.

The reflection perspective is particularly suitable for determining what is part of the environment and what is part of the self-adaptive system. This distinction is made based on the extent of control. For instance, in the robotic system, the self-adaptive system may interface with a mountable camera sensor, but since it does not manage (adapt) its functionality, the camera is considered to be part of the environment.

A *base-level subsystem* provides the system's domain functionality (i.e., application logic). For instance, in the case of robots, navigation of a robot is performed by a base-level subsystem. A base-level subsystem comprises a set of *domain models* and a set of *base-level computations*. Before we describe the meaning of these concepts, note that consistent with the reflection perspective, in FORMS we distinguish between *model*s and *computation*s. Intuitively, a model comprises *representations*, which describe something of interest in the physical and/or cyber world, while computation is an activity in a software system that manages its own states. Precise specifications of *model* and *computation* are provided using Z in Appendix A.

A *domain model* represents a domain of interest for the application logic (i.e., system's main functionality, referred to as base-level subsystem). The domain model in the robotic system may incorporate a variety of information: a map of the terrain, locations of obstacles and other robots, etc. A base-level computation perceives the environment, reasons about and acts upon a domain model, and effects the environment. As an example, consider a base-level computation of a robot dealing with battery usage. Given the current location and the remaining energy level of the battery (both are representations of the domain model), the base-level computation may select a new route, and thus change attributes of the environment.

A *reflective subsystem* is a part of the computing system that manages another subsystem, which can be either a base-level or a reflective subsystem. Note that a reflective subsystem may manage another reflective subsystem. This would be the case when a self-adaptive system includes multiple reflective levels. For instance, consider a robot that not only has the ability to adapt its navigation strategy (e.g., fastest time, minimize collisions), but also adapt the way such adaptation decisions are made (e.g., based on remaining energy level of the battery, particular environment conditions).

A *reflective subsystem* consists of two parts: *reflection model* and *reflective computation*. A reflection model reifies the entities (e.g., subsystem elements, environment attributes) needed for reasoning about adaptation. It is analogous to meta-level information in the area of computational reflection [Maes 1987]. In many self-adaptive systems, the reflection model corresponds to the software system's architectural models [Oreizy et al. 1998; Kramer and Magee 2007]. Analogous to a base-level computation, a reflective computation reasons about and acts upon reflection models. For instance, a reflection model for the robot scenario may be a component-and-connector view of the running software system, which is used at runtime by the robot's adaptation logic (i.e., reflective computation) to add/remove software components. A reflective computation also monitors the environment to determine when/if adaptations are necessary. For instance, the reflective computation in a robotic system may monitor the maneuverability complexity of a terrain to determine the best navigation component (algorithm) for execution. However, note that unlike the base-level computation, a reflective computation does not have the ability to effect changes on the environment directly. The rationale is separation of concerns (disciplined split [Maes 1987]): reflective computations are concerned with a base-level subsystem, base-level computations are concerned with a domain.

The portion of FORMS described above is inspired by the concepts from computational reflection, which as mentioned earlier have historically influenced the design of a large class of self-adaptive software systems. As demonstrated in Section 4, applying this model to any self-adaptive system naturally delineates the boundaries between various key elements of such systems. In particular, the reference model helps to clearly distinguish between elements that constitute the environment, the base-level (managed) subsystem, and the reflective (adaptation reasoning) subsystem. However, this perspective does not allow for specification of several other concerns that may arise in the architectural specification of a self-adaptive system. Next we describe how the model is extended to incorporate distribution concerns.

3.2 Unification with Distribution Perspective

The reflection perspective of FORMS provides a modularization of self-adaptive systems in the form of different layers that deal with different concerns. However, this perspective says little about the modularization within each layer. The structure within each layer is particularly important for self-adaptive systems that are distributed, as they make up the majority of real-world systems.

In a distributed setting, the software systems deployed on different nodes require dedicated *coordination mechanisms* [Ossowski and Menezes 2006] to realize the goals. In general, a coordination mechanism allows resolution of coordination problems that arise from dependencies [Malone and Crowston 1994], such as managing dependencies between multiple tasks and multiple resources.

The choice for a coordination mechanism depends on the requirements of the system and the characteristics of its environment. For example, coordination in a client-server system may be fairly easily achieved by an explicit call-return protocol. However, for other classes of distributed systems, such as ubiquitous systems [Weiser 1993] and multi-agent systems [Wooldridge and Jennings 1995] that are highly dynamic, more advanced coordination mechanisms are required, since in such systems there is typically no central point of control.

Adding self-adaptation to a distributed system similarly requires proper support for coordination of the reflective computations that deal with the adaptations. Fig. 2 shows an overview of the FORMS primitives and their relationships for the distribution perspective. The Z specification of the model is provided in Appendix B.

A *distributed self-adaptive system* consists of multiple *local self-adaptive systems*. In the

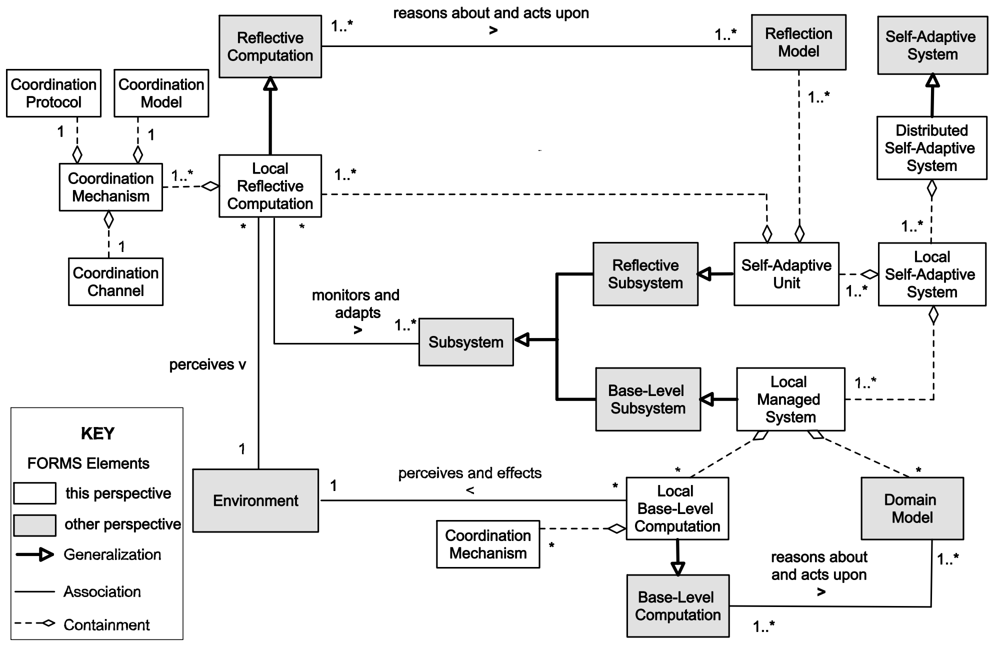


Fig. 2. Unification of FORMS primitives derived from reflection perspective with those from the distribution perspective.

robotic case, the software running on a collection of robots forms a distributed self-adaptive system, while the software deployed on each robot constitutes a local self-adaptive system. A local self-adaptive system comprises *local managed systems* and *self-adaptive units*. A local managed system provides the system's domain functionality, similar to a base-level subsystem. A local managed system comprises *domain models* and *local base-level computations*. Local base-level computation extends base-level computation with a *coordination mechanism*. In the example case, the local base-level computations of the robots may have to coordinate for collision avoidance. Such coordination may be achieved by different types of protocols. For example, in a hierarchical approach, one master robot may control the allowed movements of the slave robots. However, in a peer-to-peer approach, the robots may use a distributed locking mechanism to avoid collisions.

A self-adaptive unit manages another part of the system, which can be either one or several local managed systems or self-adaptive units, similar to a reflective subsystem. A self-adaptive unit comprises a set of *reflection models* and a set of *local reflective computations*. A local reflective computation extends a reflective computation with *coordination mechanisms* which allow the computation to coordinate with other local reflective computations in the same layer.

FORMS's *coordination mechanism* is a composite consisting of a *coordination model*, a *coordination protocol*, and a *coordination channel*. This is a commonly accepted structure for coordination mechanisms, see for example [Andrade et al. 2000] and [Arbab 2004].

A *coordination model* contains the data used by a local reflective computation to coordinate with reflective computations of other self-adaptive units. It represents information such as the current coordination partners and their roles, status information about the ongoing interactions, etc. Robots that coordinate to deal with version management may keep track of the current version of the local software running on its robot and probably other robots, the location where to download new versions, etc.

The *coordination protocol* represents the rules that govern the coordination among the participating computations. Examples of protocols are *master-slave* in an *organization-based* coordination mechanism and an *auction* in a *marked-based* coordination mechanism. As an example, robots may use *heartbeat* as a coordination protocol to detect possible failures.

A *coordination channel* is a semantic connector that acts as the means of communication between the parties involved in a coordination. A coordination channel can be an abstraction for direct interactions (regular communication channels for message exchange) as well as indirect interactions (e.g. shared tuple spaces). Heartbeat for failure detection in the robotic case may use *broadcast* as a coordination channel.

The distribution perspective emphasizes the modularization of self-adaptive system within a layer. The perspective capture, among other aspects, the degree of autonomy of the self-adaptive units, i.e. the degree to which reflective computations of a self-adaptive unit are able to make local adaptation decisions. To realize the adaptation goals, the computations of self-adaptive units in a distributed self-adaptive system have to coordinate. The high-level FORMS primitives that support coordination are based on established work in the field of coordination. Our experience shows that the distribution perspective supports the specification of a variety of distributed self-adaptive systems. We give examples in section 4 and the Appendix.

3.3 Unification with MAPE-K Perspective

One of the most commonly employed frameworks for describing and understanding self-adaptive systems is IBM's framework for Autonomic Computing [Kephart and Chess 2003; IBM 2006], which itself is inspired by the use of a feedback-control loop [Shaw 1995] in the design of software systems. The framework is formed around the notion of autonomic manager that implements a MAPE-K control loop. MAPE-K's power is its intuitive structure of the different computations that are involved in realizing the feedback-control loop in self-adaptive software systems. On the other hand, MAPE-K is neither formalized, nor does it address some of the other important self-adaptation concerns, such as those described in the previous two sections. Figure 3 shows the relationship among FORMS primitives inspired by MAPE-K with the modeling primitives from other perspectives. In the process of unifying the MAPE-K perspective with FORMS, we identified several additional primitives that are not present in MAPE-K, including a refinement of knowledge. The MAPE-K perspective is detailed in Appendix C.

As mentioned before, in FORMS a self-adaptive unit is a self-contained entity that adapts the local managed system using several *reflective computations*, which use a set of *reflection models*. The MAPE-K perspective allows us to describe the abstract notions of reflective computation and reflection model more concretely.

In FORMS, we distinguish between four types of *reflection models*: *subsystem model*, *concern model*, *environment model*, and *MAPE working model*.

A *subsystem model* represents (parts of) the *system* that is managed by the self-adaptive unit. The subsystem 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-meta-level model). In the robotic application, the architectural models representing the structure of the managed software system correspond to the subsystem model.

A *concern model* represents the objectives or goals of a self-adaptive unit. In the robotic system for example, a self-healing concern can be represented as rules of the form *event–condition–action set*. *Event* is a failure of a software component, *condition* is a local dependency on the failing component, and *action set* comprises a set of repair actions required to recover from the failure.

An *environment model* reifies the relevant part of the environment at the reflective level. In

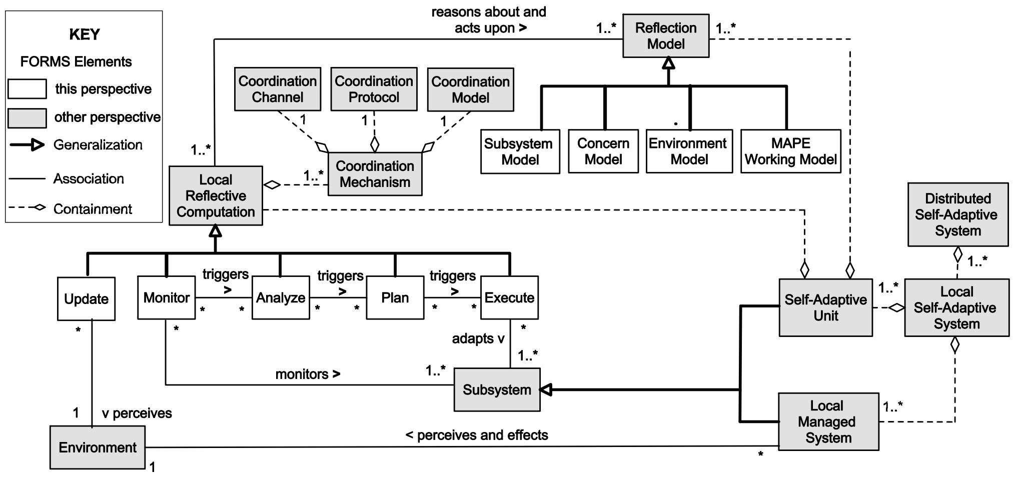


Fig. 3. FORMS reference model derived from the unification of reflection and distribution perspectives with those from the MAPE-K perspective.

the robotic example an environment model may for instance represent the physical environment, robot locations, and any other relevant environmental attributes.

A *MAPE working model* represents runtime data shared between the reflective computations. These models are typically domain-specific. Examples of working models in a robotic system are the temporary representations of candidate deployment architectures for adapting the domain logic (i.e., the base-level subsystems).

Reflective computations are the typical control loop computations found in self-adaptive systems: *monitor*, *analyze*, *plan*, and *execute*. In addition, we introduce *update* computations.

An *update* computation perceives the state in the environment and reflect this in the environment model. Update computations (in combination with analysis computations which we describe below) provide for context-awareness [Schilit et al. 1994], which is an important property in almost every self-adaptive system. In the example, a robot uses an update computation that employs a camera to update the positions of other robots in its environment model.

A *monitor* computation monitors the *subsystem* that is managed by the self-adaptive unit. The subsystem can be either a local managed system or another lower-level self-adaptive unit. Monitor uses the observed data to update the subsystem model. Additionally, it may *trigger* analyze computations when particular conditions hold. For example, in the robotic system, the monitor computation collects data from the managed system to determine failures of the software components, which trigger the adaptation process.

An *analyze* computation assesses the collected data to determine the system's ability to satisfy its objectives. Monitor and analyze computations provide for self-awareness [Hinchey and Sterritt 2006], which is a key property of self-adaptive systems. A *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 a violation of the system's objectives. In the robotic system, analysis and planning may determine a failure (based on the data collected by monitor) and find a solution for mitigating the failure through adaptation (e.g., re-instantiating a component).

Finally, triggered by plan, an *execute* computation carries out changes on the managed system. In the robotic system, this would correspond to applying the repair actions necessary to bring the managed system to a consistent state.

The MAPE computations are enhanced with support for distribution through the coordination primitives. The reference model explicitly separates coordination from computation. Each reflective computation may need to coordinate with one or more other reflective computations. The level of coordination among reflective computations determine the level of centralization in the system. In fact, the interplay of reflective computations using coordination mechanisms gives way to a variety of self-adaptation patterns.

4. APPLYING THE REFERENCE MODEL

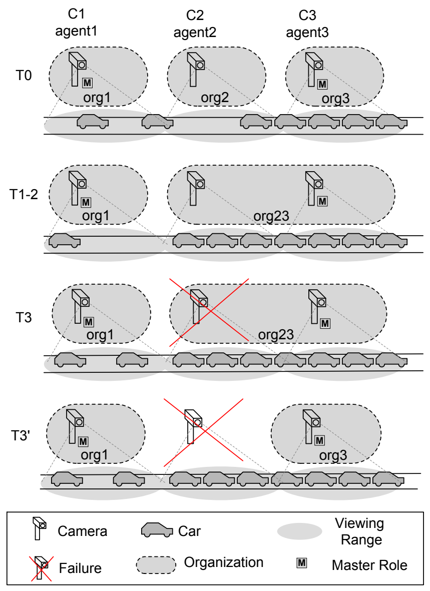
We have applied FORMS to several case studies. To that end, we describe the concepts and entities found within each case study via FORMS's high-level primitives. The purpose of the study is twofold: (1) to demonstrate the expressiveness and extensibility of the high-level reference model, and (2) to demonstrate the ability to reason about self-adaptive properties of the modeled systems. This is demonstrated for both the graphical notation and for the Z specification. In this section, we study a distributed traffic monitoring application that includes a coordination mechanism to support self-healing. Appendix E applies two FORMS perspectives to model IBM's autonomic computing framework [IBM 2006]. Finally, Appendix F uses FORMS to model a complex sensor network system called MIDAS [Malek et al. 2007], which utilizes multiple coordination mechanisms.

Traffic Monitoring System

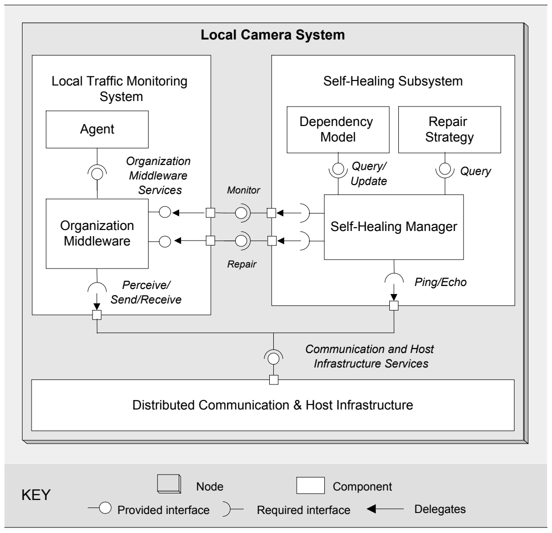
The traffic monitoring system consists of a set of intelligent cameras which are distributed evenly along the road. A simple example of a highway from this case study is shown in Figure 4(a). Each camera has a limited viewing range and cameras are placed to get an optimal coverage of the highway with a minimum overlap. The task of the cameras is to 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. 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. Such failures may bring the system to an inconsistent state and disrupt its services.

Figure 4(b) 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, i.e., the 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 collaborate with other agents to report a possible traffic jam to clients. In normal traffic conditions, each agent belongs to a single member *organization*. An example at T0 is *agent1* of organization *org1*. However, when a traffic jam is detected that spans the viewing range of multiple neighboring cameras, organizations on these cameras will merge in 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 by synchronizing with all of the slaves and masters of neighboring organizations. At T1-2, two agents, *agent2* and *agent3*, form the organization *org23*. 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*.

To support robustness to node failures, a *self-healing subsystem* is added to the system that is responsible for dealing with camera failures, as shown in Figure 4(b). A self-healing subsystem



(a) Scenario with a failing camera



(b) Deployment view of one camera

Fig. 4. Traffic monitoring case study.

comprises the following components:

- *Dependency Model* contains a model of the dependencies of the components of the local traffic monitoring system with other cameras. The *Query/Update* interface provides access for inspecting and updating the model. Dependencies include neighbor relationships, master/slave relationships, etc.
- *Repair Strategy* contains a set of *repair actions* to bring the main system to a consistent state in case a failure of a camera is detected on which this node depends. Examples of repair actions are: remove the slave of the failing camera from the list of slaves, remove the reference to the communication link with the failing camera, etc.
- *Self-Healing Manager* provides 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 sends *ping* messages to the cameras with a dependency in the dependency model. When a failure is detected (i.e., no *echo* message is received after a predefined *wait time*), the self-healing manager executes the repair actions of the repair strategy using the *Repair* interface, bringing the local traffic monitoring system back to a consistent state.

Figure 4(a) shows a failure of camera 2 at T3. The self-healing managers on the neighboring nodes will detect this after the timeout of the ping messages and then apply the repair actions. The self-healing manager on camera 1 will change its neighbor to camera 3 and visa versa, the self-healing manager on camera 3 will also remove the slave from the organization, etc. At T3' the system has recovered from the failure and can continue its correct operation. The decentralized approach for self-healing described above builds upon the MACODO model and middleware platform. The interested reader may refer to [Weyns et al. 2010b].

Figure shows the specification of the traffic monitoring case using FORMS. By extending the FORMS primitives we can precisely define the elements of the traffic monitoring system.

The *traffic environment* is refined and includes besides the attributes and processes of the

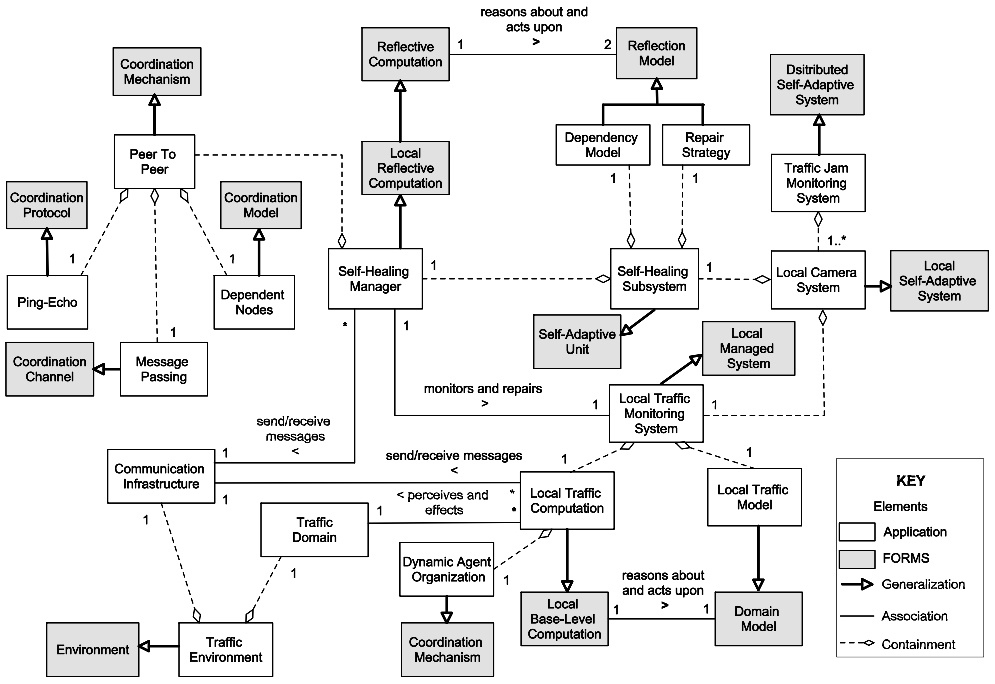


Fig. 5. Specification of traffic monitoring systems via FORMS concepts. White boxes represent traffic monitoring system constructs, gray boxes represent FORMS constructs.

traffic domain also a *communication infrastructure*. This infrastructure is used by the *local traffic computations* to coordinate the agent organizations, and by the *self-healing managers* to coordinate for failure management. A self-healing manager extends a *local reflective computation*. It uses a *peer-to-peer* coordination mechanism to deal with the failure management concern. The protocol used by self-healing managers is *ping-echo* which uses traditional *message passing* as *coordination channel*. The *dependent nodes* model maintains the list of nodes on which the local traffic monitoring system depends.

We now illustrate with excerpts of the Z specification how the FORMS model supports reasoning about the recovery of a camera in the failure scenario shown in Figure 4(a). We have kept the specification as simple as possible. Adding parameters to the Z schemes would increase their reusability, but at the cost of decreased readability. A complete Z specification of the application with the failure scenario is provided in Appendix D.

Our focus will be on the self-healing subsystem of camera 1 that detects a failure of camera 2 and adapts the local traffic monitoring system to deal with the failure.

We define an environment as a non-empty set of attributes and a set of processes that can modify the attributes:

Environment

$attributes : \mathbb{P}\, Attribute$

$processes : \mathbb{P}\, Process$

$attributes \neq \varnothing$

Changes of attributes in the environment are defined as follows:

$Change : \mathbb{P}\, Attribute \leftrightarrow \mathbb{P}\, Attribute$

Events are defined as changes generated by environment processes:

$$Event : Process \leftrightarrow Change$$

A traffic environment is defined as an environment with traffic attributes and traffic processes:

$$\begin{array}{l}
\underline{\ \mathit{TrafficEnvironment}\ }\\
\mathit{Environment}\\
\hline
\mathit{attributes} \subseteq \mathit{traffic\_domain\_attributes}\\
\mathit{processes} \subseteq \mathit{traffic\_domain\_processes}
\end{array}$$

The traffic environment at time T0 in the example (Figure 4(a)) is defined as:

$$\begin{array}{l}
\underline{\ \mathit{TrafficEnvironment}_{T0}\ }\\
\mathit{TrafficEnvironment}\\
\hline
\mathit{attributes} = \{\mathit{camera}_1\ , \mathit{camera}_2\ , \mathit{camera}_3\ , \mathit{freeflow\_zone}_1\ , \mathit{freeflow\_zone}_2\ ,\\
\qquad \mathit{congested\_zone}_3\}\\
\mathit{processes} = \mathit{traffic\_domain\_processes}
\end{array}$$

We consider the following set of events in the traffic environment:

$$\begin{array}{l}
\mathit{events} : \mathbb{P}\ \mathit{Event}\\
\hline
\mathit{events} = \{\mathit{traffic}_2 \mapsto (\{\mathit{freeflow\_zone}_2\} \mapsto \{\mathit{congested\_zone}_2\})\ ,\\
\qquad\qquad \mathit{monitor\_camera}_2 \mapsto (\{\mathit{camera}_2\} \mapsto \{\})\}
\end{array}$$

A failure of camera 2 changes the traffic environment as follows:

$$\begin{array}{l}
\underline{\ \mathit{TrafficEnvironment}_{T3}\ }\\
\Delta\, \mathit{TrafficEnvironment}_{T2}\\
p? : \mathit{Process}\\
c? : \mathit{Change}\\
s? : \mathit{shutdowns}\\
\hline
p? = \mathit{monitor\_camera}_2\\
c? = \{\mathit{camera}_2\} \mapsto \{\}\\
(p?, c?) \in \mathit{events}\\
s? = \mathit{monitor\_camera}_2\\
\mathit{attributes}' = \mathit{attributes} \setminus \mathit{first}(c?) \cup \mathit{second}(c?)\\
\mathit{processes}' = \mathit{processes} \setminus \{s?\}
\end{array}$$

The specification states that after the event, camera 2 is no longer available for traffic monitoring, and consequently, the traffic monitoring process of camera 2 is shutdown.

The domain logic of the traffic application is realized by a local traffic monitoring system deployed on each node:

$$\begin{array}{l}
\underline{\ \mathit{LocalTrafficMonitoringSystem}\ }\\
\mathit{trafficModel} : \mathit{LocalTrafficModel}\\
\mathit{computation} : \mathit{LocalTrafficComputation}\\
\hline
\mathrm{dom}\ \mathit{computation.read} = \{(\mathit{trafficModel}, \mathit{computation.state})\} \wedge\\
\mathrm{dom}\ \mathit{computation.write} = \{(\mathit{computation.state}, \mathit{trafficModel})\} \wedge\\
\mathrm{dom}\ \mathit{computation.send} = \{\mathit{computation.state}\}
\end{array}$$

A local traffic monitoring system consists of a traffic model that maintains a representation of the local traffic context and a computation that interacts with other computations to provide traffic jam monitoring services. For details we refer the reader to the Appendix. The predicate states that a local traffic computation is restricted to act upon the local traffic model, and messages for coordination are produced based on the current state of the computation.

We now zoom in on the self-healing subsystem. A dependency model is defined as a mapping of dependencies to names of cameras:

$$\begin{array}{|l}\hline \textit{DependencyModel} \\ \textit{dependencies} : \textit{Dependency} \leftrightarrow \textit{Name} \\ \hline\end{array}$$

The dependency model for camera 1 at T2 is defined:

$$\begin{array}{|l}\hline \textit{DependencyModelOne}_{T2} \\ \textit{DependencyModel} \\ \hline \textit{dependencies} = \{\textit{neighbor} \mapsto 2, \textit{neighbormaster} \mapsto 3, \textit{myslave} \mapsto 0, \textit{mymaster} \mapsto 0\} \\ \hline\end{array}$$

Camera 1 has a dependency with camera 2 as neighbor and with camera 3 as neighbor master of another organization. The number "0" indicates that camera 1 currently has no dependencies with slaves or a master within its organization.

A repair strategy model is defined as a set of repair actions:

$$\begin{array}{|l}\hline \textit{RepairStrategy} \\ \textit{repairActions} : \textit{RepairActions} \\ \hline\end{array}$$

The repair strategies for the camera 1 at time T2 is defined as:

$$\begin{array}{|l}\hline \textit{RepairStrategyOne}_{T2} \\ \textit{RepairStrategy} \\ \hline \textit{repairActions} = \{\textit{neighbor} \mapsto (2,3), \textit{neighbormaster} \mapsto (3,2)\} \\ \hline\end{array}$$

The predicate states that if camera 2 fails the new neighbor of camera 1 will be camera 3, and if camera 3 fails, camera 2 will be the master its neighbor organization.

The coordination mechanism for fault detection is defined as:

$$\begin{array}{|l}\hline \textit{PeerToPeer} \\ \textit{CoordinationMechanism}[\textit{PingEcho}, \textit{DependentNodes}, \textit{MessagePassing}] \\ \textit{pingTime} : \textit{Name} \leftrightarrow \textit{Time} \\ \textit{waitTime} : \textit{Time} \\ \hline \operatorname{dom} \textit{pingTime} = \textit{model.nodes} \wedge \forall\, n : \textit{model.nodes} \bullet \exists\, l : \textit{channel.links} \bullet \textit{first}(l) = n \\ \hline\end{array}$$

Ping time maintains the points in time when the last ping messages were sent to each of the cameras with a dependency. Wait time is a constant that indicates when an echo message should arrive after a ping messages has been sent. The last part of the predicate states that there are communication links available to each camera in the dependency model.

The concrete instance of the coordination mechanism for camera 1 at T2 is defined:

$$\begin{array}{|l}\hline \textit{PeerToPeerOne}_{T2} \\ \textit{PeerToPeer} \\ \hline \textit{model.nodes} = \{2, 3\} \\ \textit{channel.links} = \textit{traffic\_communication\_channel} \setminus \{1 \mapsto \textit{cam}_1\} \\ \textit{pingTime} = \{2 \mapsto 4430, 3 \mapsto 4440\} \\ \textit{waitTime} = 40 \\ \hline\end{array}$$

The predicate states that camera 1 has dependencies with camera 2 (its neighbor) and camera 3 (the master of its neighbor organization). The coordination mechanism has communication channels available to all the other cameras in the system. The last ping message was sent to camera 2 at time 4430 and to camera 3 at time 4440. Finally, the wait time for echo messages is 40 time units.

A self-healing manager is defined:

$$\begin{array}{|l}
\textit{SelfHealingManager} \\
\hline
\textit{Computation} \\
\textit{coordinationMechanism} : \textit{PeerToPeer} \\
\textit{read} \ldots \\
\textit{sense} : \textit{LocalTrafficMonitoringSystem} \times \mathbb{P}\,\textit{State} \rightarrow \mathbb{P}\,\textit{State} \\
\textit{adapt} : \textit{LocalTrafficMonitoringSystem} \times \mathbb{P}\,\textit{State} \rightarrow \textit{LocalTrafficMonitoringSystem} \\
\textit{send} : \mathbb{P}\,\textit{State} \rightarrow \textit{Message} \\
\textit{receive} : \textit{Message} \rightarrow \mathbb{P}\,\textit{State} \\
\hline
\end{array}$$

A self-healing manager is a computation extended with a peer-to-peer coordination mechanism. Self-healing manager can read and write a dependency model and repair actions (omitted). It can sense a local traffic monitoring system and adapt it when a failure of a dependent camera is detected. Coordination with other self-healing managers is done using the exchange of messages.

The self-healing manager of camera 1 at time T2 is defined:

$$\begin{array}{|l}
\textit{SelfHealingManagerOne}_{T2} \\
\hline
\textit{SelfHealingManager} \\
\textit{PeerToPeerOne}_{T2} \\
\hline
\end{array}$$

A self-healing subsystem is than defined as:

$$\begin{array}{|l}
\textit{SelfHealingSubsystem} \\
\hline
\textit{dependencyModel} : \textit{DependencyModel} \\
\textit{repairStrategy} : \textit{RepairStrategy} \\
\textit{selfHealingManager} : \textit{SelfHealingManager} \\
\hline
\ldots \\
\hline
\end{array}$$

A self-healing subsystem comprises a dependency model, a repair strategy and a self-healing manager. The omitted predicate defines the scope of the allowed actions of the self-healing manager.

The concrete self-healing subsystem for camera 1 at T2 is defined:

$$\begin{array}{|l}
\textit{SelfHealingSubsystemOne}_{T2} \\
\hline
\textit{SelfHealingSubsystem} \\
\textit{DependencyModelOne}_{T2} \\
\textit{RepairStrategyOne}_{T2} \\
\textit{SelfHealingManagerOne}_{T2} \\
\hline
\end{array}$$

A timeout of a self-healing manager is defined as:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Timeout}} \\
\Xi \mathit{SelfHealingManager} \\
\mathit{Tick} \\
n! : \mathit{Name} \\
\hline
\exists\, n! : \mathit{Name};\ t : \mathit{Time} \bullet (n!, t) \in \mathit{coordinationMechanism.pingTime} \wedge \\
\qquad t + \mathit{coordinationMechanism.waitTime} > \mathit{time}' \\
\hline
\end{array}$$

The schema tells us that a timeout does not change its state. A timeout happens when the clock makes a tick. The predicate states that a timeout for a particular camera is reached when the time after the tick exceeds the last ping time for that camera plus the wait time.

We now explain how self-healing is realized for one of the cameras. The timeout for self-healing manager 1 after the crash of camera 2 is defined as:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Timeout}_1} \\
\mathit{Timeout} \\
\Xi \mathit{SelfHealingManagerOne}_{T2} \\
\hline
\mathit{time} = 4470 \\
n! = 2 \\
\hline
\end{array}$$

The timeout happens when the clock makes a tick at time “4470” (recall that the ping message to camera 2 was sent at time “4430” and the waiting time is 40 time units). The timeout applies for camera 2.

Finally, the recovery of camera 1 for the failure of camera 2 is defined as:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{CameraOneRecoversFromFailureCameraTwo}} \\
\Delta\, \mathit{TrafficJamMonitoringSystem}_{T3} \\
\mathit{TrafficEnvironment}_{T3} \\
\mathit{Timeout}_1 \\
\mathit{lcs}1?, \mathit{lcs}1! : \mathit{SituatedLocalCameraSystem} \\
\mathit{camera} : \mathit{Attribute} \\
\mathit{cam} : \mathit{EnvironmentRepresentation} \\
n : \mathit{Name} \\
\hline
\{\mathit{camera}\} = \mathit{first}(c?) \wedge \\
\mathit{traffic\_communication\_channel} = \mathit{traffic\_communication\_channel} \setminus \{n \mapsto \mathit{cam}\} \wedge \\
\ldots \\
\mathit{lcs}1?.\mathit{myName} = 1 \wedge \\
\mathit{lcs}1!.\mathit{context} = \mathit{lcs}1?.\mathit{context} \setminus \{\mathit{camera}\} \wedge \\
\mathit{lcs}1!.\mathit{selfHealingSubsystem} = \mathit{updateSelfHealingSubsystem}(\mathit{lcs}1?, \mathit{camera}, \mathit{cam}, n) \wedge \\
\mathit{lcs}1!.\mathit{localTrafficMonitoringSystem} = \\
\qquad \mathit{adaptLocalTrafficMonitoringSystem}(\mathit{lcs}1?, \mathit{camera}, \mathit{cam}, n) \wedge \\
\mathit{localCamaraSystems}' = \mathit{localCamaraSystems} \setminus \{\mathit{lcs}1?\} \cup \{\mathit{lcs}1!\} \\
\hline
\end{array}$$

The specification declaratively specifies the adaptations the local camera system after the failure of the camera. The first part of the predicate selects the failing camera using the camera failure event. Next, the communication channels are updated. Then, some minor aspects are omitted. Subsequently, the recovering local camera system is selected (with myName = 1) and the failing camera is removed from its context. Finally, the adaptation is specified, consisting of two parts: an update of the state of the self-healing subsystem and the actual adaptation of the local traffic monitoring system (using two helper functions that are omitted here). From an operational point of view, the self-healing manager will update its state and apply the adaptation of the local traffic monitoring system using various read and write operations.

5. RELATED WORK

We adopt a broad perspective in the review of the related literature. We consider research from the pervasive and ubiquitous computing area as well as the research from the self-adaptive and autonomic computing area. This is reasonable given that these systems are highly related, and the ability to deal with the dynamic and unpredictable nature of ubiquitous and pervasive systems is generally considered as one of the primary motivations for autonomic computing [Sterritt 2005] and self-adaptive systems.

The main influences on the work presented herein are computational reflection, feedback-control loop pattern, and distributed coordination. We already discussed these influences, and the contribution from frameworks and reference implementations in detail in Section 3, mainly with examples from the self-adaptive and autonomic systems area. The contribution of FORMS, with respect to these frameworks and reference applications, is the integrated view aimed at encompassing different points of view represented by these and relating elements from different perspectives to one another.

We have also found support for our modeling perspectives from works within the ubiquitous and pervasive computing community. For instance, Capra et al. [Capra et al. 2001] advocate the use of reflection and meta-data in middleware to support the construction of context-aware applications. [Nahrstedt et al. 2001] describes how a control loop may be used to engineer QoS based adaptations in ubiquitous environments. Sometimes the nature of pervasive computing systems require that the control loop to be "opened", involving humans-in-the-loop [Erickson 2002]. Adding to the problems in the distribution perspective is the fact that in a ubiquitous system the distributed processes may run on mobile devices [Fok et al. 2004] and in some instances the processes themselves are mobile [Carzaniga et al. 1997]. This calls for dedicated techniques, in particular to enable coordination [Braione and Picco 2004; Murphy et al. 2006]. A problem is the lack of a, coherent, unifying model, with support for all three perspectives and formal underpinnings that provide for the required precision and expressibility in industrial scale software development projects.

Several formal approaches targeting specific aspects of self-adaptation exist. For instance, [Zhang and Cheng 2006] present an approach to formally model the behavior of adaptive programs, automatically analyze them, and generate an implementation of the system. [Wermelinger and Fiadeiro 1999] presents an algebra for formally specifying runtime reconfigurations of a system's software architecture. Several formal approaches also target pervasive and context-aware computing systems, e.g., process calculus approaches such as mobile ambients [Cardelli and Gordon 2000]. ASSL [Vassev and Hinchey] provides support for a complete development methodology of self-adaptive embedded systems, including specification and verification of self-adaption. These and other works demonstrate the usefulness of applying formal modeling to this field. In comparison, FORMS provides an encompassing formally founded vocabulary for describing and reasoning about different concerns of self-adaptive software architectures which is a different, complementary focus.

Another challenge for these systems is to model key system aspects, for instance the environment and how it is perceived. In the pervasive and context aware domain, an important focus has been on specifying, interpreting, recognizing and storing contextual information [Ranganathan and Campbell 2003; Dey 2000; Henricksen et al. 2002; Román et al. 2002] using formal or semi-formal models. [Schmidt et al. 1999] proposes a layered architecture with formal underpinnings for sensor based context recognition. [Brewington and Cybenko 2000] discuss how to manage context monitoring when context information is transient. They provide a formal reasoning framework for deciding when to update context information via the system's sensors. This is an example of extended semantic description of the monitor concept in the FORMS MAPE-K perspective. The semantic web has influenced several ontology based approaches [Román et al. 2002; Ye et al. 2007], not just for modeling context, but also other critical system aspects such as trust [Haque and Ahamed 2007] and even coordination of application invocation [Román et al. 2002].

6. DISCUSSION

The application of FORMS to the traffic monitoring system demonstrates the expressive power and extensibility of the FORMS primitives. It also shows how the specification allows reasoning about a self-healing property of the system at the architectural level. The step-wise analysis and refinement of the specification starting with the event of the failing camera up to the recovery of the camera highlights the key design elements of this self-healing scenario. Describing such reasoning steps can be a useful way to get better insight in the self-adaptive property, to detect problems at early stages of system construction, or it can serve as documentation for detailed design and system implementation. Beyond specification and reasoning about the high-level architectural design of a self-adaptive system, the formalism affords numerous capabilities. For instance, existing tools could be used for: (1) *type checking* (e.g., CZT's type checker [CZT 2010]) to automatically obtain certain guarantees on the validity of the specification of a self-adaptive system, such as conformance of architecture descriptions that are refined iteratively, (2) *executing* and *animating* the schemas (e.g., CZT's ZLive animator [CZT 2010]) to visually obtain a better understanding of the system's properties, and (3) *testing* (e.g., CZT's ModelJUnit [CZT 2010]) to automatically generate test cases.

In light of the above discussion, the contributions of FORMS can be summarized as follows. First, FORMS establishes a shared vocabulary of primitives in this area that while simple and concise can be used to precisely describe the essential aspects of complex self-adaptive systems. Second, FORMS enables engineers to specialize the primitives for their specific domain and concerns of interest. Third, it enables engineers to reason about their early design decisions, which are known to be the most difficult to make but have the most impact on system construction and evolution. Finally FORMS lays a foundation for a systematic method of developing a pattern catalog of known solutions (i.e., architectural patterns).

However, the formal reference model is not without limitations. First, the primitives of FORMS are coarse-grained. While the abstractions cover a wide variety of domains, from our experiences we learned that in most cases the primitives need to be refined to be really useful for an engineer. Second, reasoning about the description of a self-adaptive system is most appropriate with a specification in Z. However, this implies that the engineer is familiar with Z in general and the specification of the FORMS perspectives in particular. Moreover, such specifications tend to be lengthy. Third, while excellent tools are available for the specification of a system in Z, less support is available for reasoning about the system and automatic verification of properties. Finally, currently, FORMS does not support consistency and trace-ability between a specification and an implementation. While such support would be attractive from a practical point of view, it was clearly not in the scope of the research presented in this paper.

7. CONCLUSIONS AND FUTURE WORK

The emergence of pervasive and ubiquitous computing environments, which are often highly dynamic and unpredictable, have motivated the development of self-adaptive software systems. However, building self-adaptive software systems has shown to be significantly more challenging than traditional software systems. There are numerous technical culprits, but we believe the one that hinders progress the most is the lack of a precise vocabulary for describing and reasoning about the primary architectural characteristics of self-adaptive systems.

This is exactly the challenge we have undertaken in this paper. We have presented FORMS, a formal reference model for specifying self-adaptive software systems. Unlike existing guidelines and frameworks proposed previously, FORMS aims to incorporate various points of view into a unifying reference model. We presented unification of three perspective that have historically influenced the majority of existing approaches employed in the construction of self-adaptive systems: computational reflection, distributed coordination, and MAPE-K. We distilled the self-adaptation primitives from these three perspectives and related them to one another, and provided a formal definition of them using Z notation. We have demonstrated FORMS precision, expressiveness, and extensibility by applying it to several case studies.

While our experiences with FORMS have been very positive, several avenues of future work remain. We intend to investigate new concerns in self-adaptation to further assess, and potentially extend, FORMS's perspectives. The formally defined reference model primitives form the basis for a design language. A language we plan to use for documenting architectural patterns in this setting. In turn, by studying the relationships between patterns and their quality attributes, we intend to develop a catalog of reusable strategies and tactics for building systems in this area.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

REFERENCES

ANDERSSON, J. ET AL. 2009a. *Modeling dimensions of self-adaptive software systems*. In B. H. C. Cheng et al., editors, LNCS vol. 5525, Hot Topics on Software Engineering for Self-Adaptive Systems.

ANDERSSON, J. ET AL. 2009b. Reflecting on self-adaptive software systems. In *Workshop on Software Engineering for Adaptive and Self-Managing Systems*. Vancouver, BC.

ANDRADE, L. F. ET AL. 2000. Patterns for coordination. In *Int'l Conf. Coordination Languages and Models*. LNCS, vol. 1906. Springer, 317–322.

ARBAB, F. 2004. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science 14,* 3, 329–366.

BLAIR, G. ET AL. 2004. Research directions in reflective middleware: the lancaster experience. In *3rd Workshop on Adaptive and Reflective Middleware*. ARM '04. ACM.

BRAIONE, P. AND PICCO, G. P. 2004. On calculi for context-aware coordination. In *Int'l Conf. on Coordination Models and Languages*. LNCS, vol. 2949. Springer, 38–54.

BREWINGTON, B. AND CYBENKO, G. 2000. Keeping up with the changing web. *Computer 33,* 5, 52 –58.

CAPRA, L. ET AL. 2001. Reflective middleware solutions for context-aware applications. In *Int'l Conf. on Metalevel Architectures and Separation of Crosscutting Concerns*. 126–133.

CARDELLI, L. AND GORDON, A. D. 2000. Mobile ambients. *Theoretical Computer Science 240,* 1, 177–213.

CARZANIGA, A. ET AL. 1997. Designing distributed applications with mobile code paradigms. In *Int'l conf. on Software engineering*. ACM, New York, NY, USA, 22–32.

CAZZOLA, W. ET AL. 1999. Rule-based strategic reflection: Observing and modifying behavior at the architectural level. *Int'l Conf. on Automated Software Engineering*.

CHENG, B. ET AL. 2009. *Software engineering for self-adaptive systems: A research road map*. In B. H. C. Cheng et al., editors, LNCS vol. 5525, Hot Topics on Software Engineering for Self-Adaptive Systems.

CZT. 2010. http://czt.sourceforge.net/.

DEY, A. 2000. Providing architectural support for building context-aware applications. Ph.D. thesis, Atlanta, USA.

DOWLING, J. AND CAHILL, V. 2001. The k-Component architecture meta-model for self-adaptive software. In Int'l Conf. on Metalevel Architectures and Separation of Crosscutting Concerns. London, UK.

EDWARDS, G. ET AL. 2009. Architecture-driven self-adaptation and self-management in robotics systems. In *Int'l Workshop on Software Engineering for Adaptive and Self-Managing Systems*. Vancouver, BC.

ERICKSON, T. 2002. Some problems with the notion of context-aware computing. *Commun. ACM 45,* 2, 102–104.

FOK, C.-L. ET AL. 2004. A lightweight coordination middleware for mobile computing. In *COORDINATION*, R. D. Nicola, G. L. Ferrari, and G. Meredith, Eds. LNCS, vol. 2949. Springer, 135–151.

GARLAN, D. ET AL. 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*.

GEIHS, K. ET AL. 2009. Software engineering for self-adaptive systems. Springer-Verlag, Berlin, Heidelberg, Chapter Modeling of Context-Aware Self-Adaptive Applications in Ubiquitous and Service-Oriented Environments.

HAQUE, M. AND AHAMED, S. 2007. An omnipresent formal trust model (ftm) for pervasive computing environment. In *Int'l Computer Software and Applications Conf.* IEEE, Washington, DC, USA, 49–56.

HENRICKSEN, K. ET AL. 2002. Modeling context information in pervasive computing systems. In *Pervasive*, F. Mattern and M. Naghshineh, Eds. LNCS, vol. 2414. Springer, 167–180.

HINCHEY, M. G. AND STERRITT, R. 2006. Self-managing software. *Computer 39*, 107–.

HUEBSCHER, M. C. AND MCCANN., J. A. 2008. A survey of autonomic computing - degrees, models, and applications. *ACM Computing Survey 40,* 3.

IBM. 2006. An architectural blueprint for autonomic computing. Tech. rep., IBM. Jan.

KEPHART, J. O. AND CHESS, D. M. 2003. The vision of autonomic computing. IEEE Computer *36,* 1, 41–50.

KRAMER, J. AND MAGEE, J. 2007. Self-managed systems: an architectural challenge. In Int'l Conf. on Software Engineering. Minneapolis, MN.

MAES, P. 1987. Concepts and experiments in computational reflection. In OOPSLA. Orlando, FL.

MALEK, S. ET AL. 2007. Reconceptualizing a family of heterogeneous embedded systems via explicit architectural support. In *Int'l conf. on Software Engineering*. Washington, DC, USA, 591–601.

MALONE, T. AND CROWSTON, K. 1994. Toward an interdisciplinary theory of coordination. *ACM Computing Surveys 26,* 1, 87–119.

MILLER, B. 2005. The autonomic computing edge: The role of knowledge in autonomic systems. Tech. rep., IBM.

MURPHY, A. ET AL. 2006. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol. 15,* 3, 279–328.

NAHRSTEDT, K. ET AL. 2001. Qos-aware middleware for ubiquitous and heterogeneous environments. *IEEE Communications Magazine 39,* 11, 140 –148.

OREIZY, P. ET AL. 1998. Architecture-based runtime software evolution. In Int'l Conf. on Software engineering. Kyoto, Japan.

OSSOWSKI, S. AND MENEZES, R. 2006. On coordination and its significance to distributed and multi-agent systems: Research articles. *Concurr. Comput. : Pract. Exper. 18,* 4, 359–370.

RANGANATHAN, A. AND CAMPBELL, R. H. 2003. An infrastructure for context-awareness based on first order logic. *Personal Ubiquitous Comput. 7,* 6, 353–364.

ROMÁN, M. ET AL. 2002. A middleware infrastructure for active spaces. *IEEE Pervasive Computing 1,* 4, 74–83.

SCHILIT, B. ET AL. 1994. Context-aware computing applications. In *First Workshop on Mobile Computing Systems and Applications*. IEEE Computer Society, Washington, DC, USA, 85–90.

SCHMIDT, A. ET AL. 1999. Advanced interaction in context. *LNCS 1707*, 89–101.

SHAW, M. 1995. Beyond objects: A software design paradigm based on process control. ACM SIGSOFT Software Engineering Notes *20,* 1, 27–38.

STERRITT, R. 2005. Autonomic computing. *ISSE 1,* 1, 79–88.

TISATO, F. ET AL. 2001. Architectural reflection: Realising software architectures via reflective activities. In *Int'l Workshop on Engineering Distributed Objects*. Springer.

VASSEV, E. AND HINCHEY, M. The assl approach to specifying self-managing embedded systems. *Concurrency and Computation: Practice and Experience*.

WEISER, M. 1993. Ubiquitous computing. *Computer 26*, 71–72.

WERMELINGER, M. AND FIADEIRO, J. L. 1999. Algebraic software architecture reconfiguration. In European Software Engineering Conf. and Int'l Symp. on Foundations of Software Engineering. Toulouse, France.

WEYNS, D. ET AL. 2010a. On decentralized self-adaptation: Lessons from the trenches and challenges for the future. In *Int'l Workshop on Software Engineering for Adaptive and Self-Managing Systems*. Cape Town.

WEYNS, D. ET AL. 2010b. The MACODO middleware for context-driven dynamic agent organizations. *TAAS 5,* 1, 3.1–3.29.

WEYNS, D., MALEK, S., AND ANDERSSON, J. 2010a. FORMS: A formal reference model for self-adaptation. In Int'l Conf. on Autonomic Computing and Communications. Washington, DC.

WEYNS, D., MALEK, S., AND ANDERSSON, J. 2010b. Z specifications of FORMS. Tech. rep., *CW 579, K.U.Leuven*, www.cs.kuleuven.be/publicaties/rapporten/cw/CW579.abs.html.

WOOLDRIDGE, M. AND JENNINGS, N. 1995. Intelligent agents: Theory and practice. *The Knowledge Engineering Review 10,* 02, 115–152.

YE, J. ET AL. 2007. Ontology-based models in pervasive computing systems. *The Knowledge Engineering Review 22,* 04, 315–347.

ZHANG, J. AND CHENG, B. H. C. 2006. Model-based development of dynamically adaptive software. In Int'l Conf. on Software Engineering. Shanghai, China.

This document is the appendix to:

A Unifying Reference Model for Formal Specification of Distributed Self-Adaptive Software Systems

Danny Weyns, Linnaeus University, Sweden
Sam Malek, George Mason University, USA
Jesper Andersson, Linnaeus University, Sweden

In this appendix, we first give a complete formal definition of FORMS in the Z language. Subsequently, we present the reflection perspective (in section A), the unification with the distribution perspective (in section B), and the unification with the MAPE-K perspective (in section C). For each part of the formal model, we give a graphical overview of the specified elements and relations, followed by the formal specification. Next, we give a complete formal specification of a traffic monitoring example (in section D). In the last part of the appendix, we discuss two additional case studies: IBM's autonomic computing framework [IBM 2006] (in section E), and a complex sensor network system called MIDAS [Malek et al. 2007] (in section F). The complete Z specification is type checked using CZT tools [CZT 2010].

A. REFLECTION PERSPECTIVE

Fig. 6 shows a graphical overview of the FORMS elements and relations from the reflection perspective.

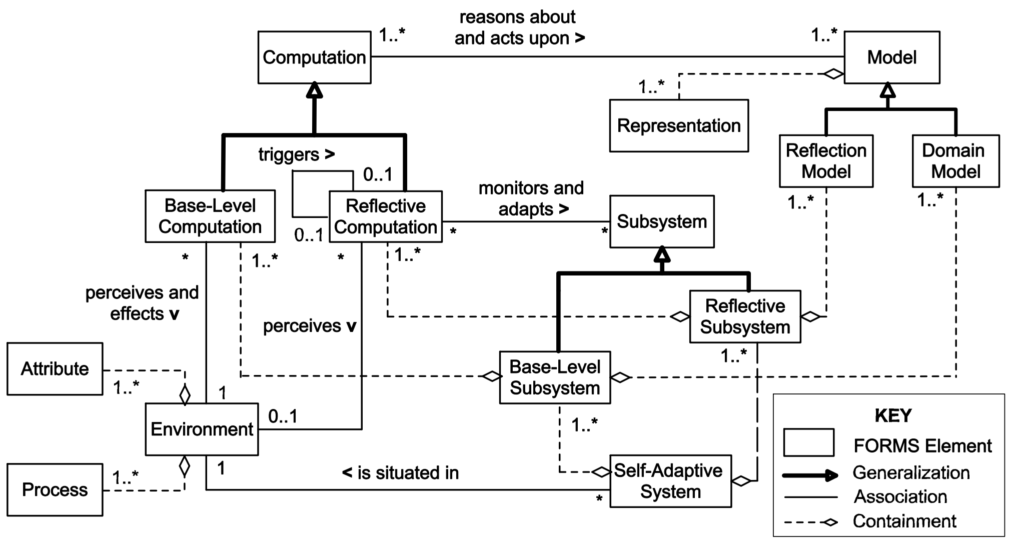


Fig. 6. FORMS: reflection perspective.

A.1 Environment

To define environment, we first introduce attributes and processes. An attribute is a perceivable characteristic of the environment. The set of attributes is defined:

$[Attribute]$

A process is an activity in the environment that can change attributes. The set of processes is defined as:

$[Process]$

An environment comprises a non-empty set of attributes and a set of processes that can modify the attributes. Environment is defined:

$$
\begin{array}{|l}
\textit{Environment} \\
\hline
attributes : \mathbb{P}\, Attribute \\
processes : \mathbb{P}\, Process \\
\hline
attributes \neq \varnothing \\
\hline
\end{array}
$$

We define context as a set of accessible attributes of the environment:

$Context == \mathbb{P}\, Attribute$

Changes in the environment are defined as follows:

$$\mid\ Change : \mathbb{P}\, Attribute \leftrightarrow \mathbb{P}\, Attribute$$

Events are defined as changes generated by processes:

$$\mid\ Event : Process \leftrightarrow Change$$

A.2 Base-Level Subsystem

A base-level subsystem provides the system's domain functionality, i.e., application logic. To define a base-level subsystem, we first introduce models. A model comprises representations that describes something of interest in the physical world and/or cyber world. Models are defined:

$$
\begin{array}{|l}
\textit{Model}\ [Representation] \\
\hline
representations : \mathbb{P}\, Representation \\
\hline
representations \neq \varnothing \\
\hline
\end{array}
$$

Representation is defined as a parameter to allow concrete models having different types of representations.

An environment representation is a representation of attributes in the environment. The set of environment representations is defined:

$[EnvironmentRepresentation]$

A domain model describes a domain of interest for one or more stakeholders. Domain model is defined:

$$\begin{array}{|l}
\textit{DomainModel} \\
\hline
\textit{Environment} \\
\textit{Model}[\textit{EnvironmentRepresentation}] \\
\textit{mapping} : \mathbb{P}\,\textit{Attribute} \leftrightarrow \textit{EnvironmentRepresentation} \\
\hline
\operatorname{dom} \textit{mapping} \subseteq \{\textit{attrs} : \mathbb{P}\,\textit{Attribute} \mid \textit{attrs} \subseteq \textit{attributes}\} \\
\operatorname{ran} \textit{mapping} = \{r : \textit{EnvironmentRepresentation} \mid \\
\qquad r \in \textit{representations}\} \\
\hline
\end{array}$$

A domain model maps representations to attribute sets.

To define computations, we introduce the type state. State represents the current status of a computation and is defined:

$[\textit{State}]$

A computation is an activity in a software system that manages its own state. Computations are defined:

$$\begin{array}{|l}
\textit{Computation} \\
\hline
\textit{state} : \mathbb{P}\,\textit{State} \\
\textit{compute} : \mathbb{P}\,\textit{State} \rightarrow \mathbb{P}\,\textit{State} \\
\hline
\operatorname{dom} \textit{compute} = \{s : \mathbb{P}\,\textit{State} \mid s \subseteq \textit{state}\} \\
\hline
\end{array}$$

The computation operation is defined as:

$$\begin{array}{|l}
\textit{ComputationOp} \\
\hline
\Delta \textit{Computation} \\
s?, s! : \mathbb{P}\,\textit{State} \\
\hline
s! = \textit{compute}(s?) \wedge \\
\textit{state}' = \textit{state} \setminus s? \cup s! \\
\hline
\end{array}$$

A base-level computation can act upon a set of domain models and can perceive a context in the environment and effect this context.

$$\begin{array}{|l}
\textit{BaseLevelComputation} \\
\hline
\textit{Computation} \\
\textit{read} : \mathbb{P}\,\textit{DomainModel} \times \mathbb{P}\,\textit{State} \rightarrow \mathbb{P}\,\textit{State} \\
\textit{write} : \mathbb{P}\,\textit{State} \times \mathbb{P}\,\textit{DomainModel} \rightarrow \mathbb{P}\,\textit{DomainModel} \\
\textit{perceive} : \mathbb{P}\,\textit{State} \times \textit{Context} \rightarrow \mathbb{P}\,\textit{State} \\
\textit{effect} : \mathbb{P}\,\textit{State} \times \textit{Context} \rightarrow \textit{Context} \\
\hline
\end{array}$$

A base-level subsystem is a software system that provides some functionality for a stakeholder or set of stakeholders. Base-level subsystem is defined:

$$
\begin{array}{l}
\textit{BaseLevelSubsystem} \\
\hline
\textit{models} : \mathbb{P}\,\textit{DomainModel} \\
\textit{computations} : \mathbb{P}\,\textit{BaseLevelComputation} \\
\hline
\forall\, c : \textit{computations} \bullet \\
\quad \operatorname{dom} c.\textit{read} = \{\textit{mdls} : \mathbb{P}\,\textit{DomainModel} \mid \textit{mdls} \subseteq \textit{models} \bullet \\
\qquad (\textit{mdls}, c.\textit{state})\} \land \\
\quad \operatorname{dom} c.\textit{write} = \{\textit{mdls} : \mathbb{P}\,\textit{DomainModel} \mid \textit{mdls} \subseteq \textit{models} \bullet \\
\qquad (c.\textit{state}, \textit{mdls})\}
\end{array}
$$

A base-level subsystem comprises a set of domain models and a set of base-level computations. The computations can act upon the domain models.

The read operation defines how a base-level subsystem computation reads a set of domain models and updates its state:

$$
\begin{array}{l}
\textit{ReadOp} \\
\hline
\Delta \textit{BaseLevelSubsystem} \\
c?, c! : \textit{BaseLevelComputation} \\
\textit{ms}? : \mathbb{P}\,\textit{DomainModel} \\
\hline
c? \in \textit{computations} \land \\
\textit{ms}? \subseteq \textit{models} \land \\
c!.\textit{state} = c?.\textit{read}(\textit{ms}?, c?.\textit{state}) \land \\
c!.\textit{compute} = c?.\textit{compute} \land \\
\textit{models}' = \textit{models} \land \\
\textit{computations}' = \textit{computations} \setminus \{c?\} \cup \{c!\}
\end{array}
$$

The compute operation defines how a base-level subsystem computation performs a computation on its state:

$$
\begin{array}{l}
\textit{ComputeOp} \\
\hline
\Delta \textit{BaseLevelSubsystem} \\
c?, c! : \textit{BaseLevelComputation} \\
s! : \mathbb{P}\,\textit{State} \\
\hline
c? \in \textit{computations} \land \\
s! = c?.\textit{compute}(c?.\textit{state}) \\
c!.\textit{state} = s! \land \\
c!.\textit{compute} = c?.\textit{compute} \land \\
\textit{models}' = \textit{models} \land \\
\textit{computations}' = \textit{computations} \setminus \{c?\} \cup \{c!\}
\end{array}
$$

The write operation defines how a base level computation acts upon a set of domain models:

$$
\begin{array}{l}
\textit{WriteOp} \\
\hline
\Delta \textit{BaseLevelSubsystem} \\
c? : \textit{BaseLevelComputation} \\
ms? : \mathbb{P}\ \textit{DomainModel} \\
ms! : \mathbb{P}\ \textit{DomainModel} \\
\hline
c? \in \textit{computations} \land \\
ms? \subseteq \textit{models} \land \\
ms! = c?.\textit{write}(c?.\textit{state}, ms?) \land \\
\textit{models}' = \textit{models} \setminus ms? \cup ms! \land \\
\textit{computations}' = \textit{computations}
\end{array}
$$

A.3 Reflective Subsystem

A reflective subsystem is a part of the computing system that manages another part of it, which can be either a base-level or a reflective subsystem. Note that a reflective subsystem may manage another reflective subsystem. This would be the case when a self-adaptive system includes multiple reflective levels. To define a reflective subsystem, we first introduce reflection models and reflective computations.

A reflection model representation reifies the entities (e.g., subsystem constructs, environment attributes) needed for reasoning about adaptation. It is analogous to meta-level information from the domain of computational reflection [Maes 1987]. A self-adaptive system has a set of reflection model representations:

$[\textit{ReflectionModelRepresentation}]$

A reflection model comprises reflection model representations:

$$
\begin{array}{l}
\textit{ReflectionModel} \\
\hline
\textit{Model}[\textit{ReflectionModelRepresentation}]
\end{array}
$$

Reflection models are used by reflective computations.
A reflective computation is defined:

$$
\begin{array}{l}
\textit{ReflectiveComputation}\ [\textit{Subsystem}] \\
\hline
\textit{Computation} \\
\textit{read} : \mathbb{P}\ \textit{ReflectionModel} \times \mathbb{P}\ \textit{State} \rightarrow \mathbb{P}\ \textit{State} \\
\textit{write} : \mathbb{P}\ \textit{State} \times \mathbb{P}\ \textit{ReflectionModel} \rightarrow \mathbb{P}\ \textit{ReflectionModel} \\
\textit{perceive} : \textit{Context} \times \mathbb{P}\ \textit{State} \rightarrow \mathbb{P}\ \textit{State} \\
\textit{sense} : \mathbb{P}\ \textit{Subsystem} \times \mathbb{P}\ \textit{State} \rightarrow \mathbb{P}\ \textit{State} \\
\textit{adapt} : \mathbb{P}\ \textit{Subsystem} \times \mathbb{P}\ \textit{State} \rightarrow \mathbb{P}\ \textit{Subsystem} \\
\textit{trigger} : \mathbb{P}\ \textit{State} \times \mathbb{P}\ \textit{ReflectiveComputation}[\textit{Subsystem}] \rightarrow \\
\qquad \mathbb{P}\ \textit{ReflectiveComputation}[\textit{Subsystem}]
\end{array}
$$

A reflective computation reasons and acts upon a subset of reflection models by reading from, and writing to the models. It also perceives certain environmental context. However, note that unlike a base-level computation, a reflective computation does not effect changes in the environment. Moreover, reflective computation not only senses (monitors) and adapts the subsystem, but also triggers other reflective computations.

A reflective subsystem is composed of reflection models and reflective computations. This is formally specified as follows:

$$
\begin{array}{l}
\textit{ReflectiveSubsystem}\ [\textit{Subsystem}] \\
\hline
\textit{models} : \mathbb{P}\ \textit{ReflectionModel} \\
\textit{computations} : \mathbb{P}\ \textit{ReflectiveComputation}[\textit{Subsystem}] \\
\hline
\forall\ c : \textit{computations} \bullet \\
\quad \operatorname{dom} c.\textit{read} = \{\textit{mdls} : \mathbb{P}\ \textit{ReflectionModel} \mid \textit{mdls} \subseteq \textit{models} \bullet (\textit{mdls}, c.\textit{state})\} \wedge \\
\quad \operatorname{dom} c.\textit{write} = \{\textit{mdls} : \mathbb{P}\ \textit{ReflectionModel} \mid \textit{mdls} \subseteq \textit{models} \bullet (c.\textit{state}, \textit{mdls})\} \wedge \\
\quad \operatorname{dom} c.\textit{trigger} = \{\textit{ct} : \mathbb{P}\ \textit{ReflectiveComputation}[\textit{Subsystem}] \mid \\
\qquad \textit{ct} \subseteq \textit{computations} \setminus \{c\} \bullet (c.\textit{state}, \textit{ct})\}
\end{array}
$$

A.4 Self-Adaptive System

The definition of self-adaptive system in the reflection perspective naturally delineates the boundaries between various key elements of such systems. In particular, the specification clearly distinguishes between elements that constitute the base-level (managed) subsystem, the reflective (adaptation reasoning) subsystem, and the environment in which the self-adaptive is situated (external to the self-adaptive system).

A self-adaptive system comprises a set of base-level and reflective subsystems. As an example, we consider a self-adaptive system with two reflective levels. We model a meta-level subsystem (i.e. a reflective systems on top of a base-level subsystem) as follows:

$\textit{MetaLevelSubsystem} == \textit{ReflectiveSubsystem}[\textit{BaseLevelSubsystem}]$

Similarly, a meta-meta-level subsystem can be defined:

$\textit{MetaMetaLevelSubsystem} == \textit{ReflectiveSubsystem}[\textit{MetaLevelSubsystem}]$

We can now model the self-adaptive system as follows:

$$
\begin{array}{l}
\textit{SelfAdaptiveSystem} \\
\hline
\textit{baseLevelSubsystems} : \mathbb{P}\ \textit{BaseLevelSubsystem} \\
\textit{metaLevelSubsystems} : \mathbb{P}\ \textit{MetaLevelSubsystem} \\
\textit{metaMetaLevelSubsystems} : \mathbb{P}\ \textit{MetaMetaLevelSubsystem} \\
\hline
\#\textit{baseLevelSubsystems} \geq 1 \\
\#\textit{metaLevelSubsystems} \geq 1 \\
\#\textit{metaMetaLevelSubsystems} \geq 1 \\
\forall\ \textit{mls} : \textit{metaLevelSubsystems};\ \textit{cm}, \textit{ce} : \textit{ReflectiveComputation} \bullet \\
\quad \textit{cm} \in \textit{mls}.\textit{computations} \wedge \textit{ce} \in \textit{mls}.\textit{computations} \wedge \\
\quad \operatorname{dom} \textit{cm}.\textit{sense} = \{\textit{bls} : \mathbb{P}\ \textit{BaseLevelSubsystem} \mid \\
\qquad \textit{bls} \subseteq \textit{baseLevelSubsystems} \bullet (\textit{bls}, \textit{cm}.\textit{state})\} \wedge \\
\quad \operatorname{dom} \textit{ce}.\textit{adapt} = \{\textit{bls} : \mathbb{P}\ \textit{BaseLevelSubsystem} \mid \\
\qquad \textit{bls} \subseteq \textit{baseLevelSubsystems} \bullet (\textit{bls}, \textit{cm}.\textit{state})\} \\
\forall\ \textit{mmls} : \textit{metaMetaLevelSubsystems}; \\
\textit{cm}, \textit{ce} : \textit{ReflectiveComputation} \bullet \\
\quad \textit{cm} \in \textit{mmls}.\textit{computations} \wedge \textit{ce} \in \textit{mmls}.\textit{computations} \wedge \\
\quad \operatorname{dom} \textit{cm}.\textit{sense} = \{\textit{mls} : \mathbb{P}\ \textit{MetaLevelSubsystem} \mid \\
\qquad \textit{mls} \subseteq \textit{metaLevelSubsystems} \bullet (\textit{mls}, \textit{cm}.\textit{state})\} \wedge \\
\quad \operatorname{dom} \textit{ce}.\textit{adapt} = \{\textit{mls} : \mathbb{P}\ \textit{MetaLevelSubsystem} \mid \\
\qquad \textit{mls} \subseteq \textit{metaLevelSubsystems} \bullet (\textit{mls}, \textit{ce}.\textit{state})\}
\end{array}
$$

The specification states that meta-level subsystems can sense and adapt base-level subsystems, while meta-meta-level subsystems can sense and adapt meta-level subsystems.

A self-adaptive system situated in an environment is specified:

$$
\begin{array}{|l}
\textit{SituatedSelfAdaptiveSystem} \\
\hline
\textit{Environment} \\
\textit{SelfAdaptiveSystem} \\
\textit{context} : \textit{Context} \\
\hline
\textit{context} \subseteq \textit{attributes} \\
\forall\, \textit{bls} : \textit{baseLevelSubsystems};\ c : \textit{BaseLevelComputation} \bullet \\
\quad c \in \textit{bls}.\textit{computations} \land \\
\quad \operatorname{dom} c.\textit{perceive} = \{\textit{attrs} : \textit{Context} \mid \\
\qquad \textit{attrs} \subseteq \textit{context} \bullet (c.\textit{state}, \textit{attrs})\} \land \\
\quad \operatorname{dom} c.\textit{effect} = \{\textit{attrs} : \textit{Context} \mid \\
\qquad \textit{attrs} \subseteq \textit{context} \bullet (c.\textit{state}, \textit{attrs})\} \\
\forall\, \textit{mls} : \textit{metaLevelSubsystems};\ \textit{cu} : \textit{ReflectiveComputation} \bullet \\
\quad \textit{cu} \in \textit{mls}.\textit{computations} \land \\
\quad \operatorname{dom} \textit{cu}.\textit{perceive} = \{\textit{attrs} : \textit{Context} \mid \\
\qquad \textit{attrs} \subseteq \textit{context} \bullet (\textit{attrs}, \textit{cu}.\textit{state})\} \\
\forall\, \textit{mmls} : \textit{metaMetaLevelSubsystems}; \\
\textit{cu} : \textit{ReflectiveComputation} \bullet \\
\quad \textit{cu} \in \textit{mmls}.\textit{computations} \land \\
\quad \operatorname{dom} \textit{cu}.\textit{perceive} = \{\textit{attrs} : \textit{Context} \mid \\
\qquad \textit{attrs} \subseteq \textit{context} \bullet (\textit{attrs}, \textit{cu}.\textit{state})\}
\end{array}
$$

The specification states that base-level subsystems can perceive and effect the context in which the self-adaptive system is situated, while reflective subsystems can only perceive the context.

Finally, we can now formally specify how a meta-level subsystem adapts a base-level subsystem:

$$
\begin{array}{|l}
\textit{MetaLevelAdaptationOp} \\
\hline
\Delta \textit{SituatedSelfAdaptiveSystem} \\
\Xi \textit{Environment} \\
\textit{rc}? : \textit{ReflectiveComputation}[\textit{BaseLevelSubsystem}] \\
\textit{bls}?, \textit{bls}! : \textit{BaseLevelSubsystem} \\
\textit{mls}?, \textit{mls}! : \textit{MetaLevelSubsystem} \\
\hline
\textit{bls}? \in \textit{baseLevelSubsystems} \land \\
\textit{mls}? \in \textit{metaLevelSubsystems} \land \\
\textit{rc}? \in \textit{mls}?.\textit{computations} \land \\
\{\textit{bls}!\} = \textit{rc}?.\textit{adapt}(\{\textit{bls}?\}, \textit{rc}?.\textit{state}) \land \\
\textit{baseLevelSubsystems}' = \textit{baseLevelSubsystems} \setminus \{\textit{bls}?\} \cup \{\textit{bls}!\} \\
\textit{metaLevelSubsystems}' = \textit{metaLevelSubsystems} \\
\textit{metaMetaLevelSubsystems}' = \textit{metaMetaLevelSubsystems}
\end{array}
$$

The specification states that self-adaptation changes the self-adaptive system, but does not effect the environment. The adaptation is performed by one of the meta-level reflective computations (*rc?*) which adapts one or more base-level subsystems.

B. UNIFICATION WITH DISTRIBUTION PERSPECTIVE

Fig. 7 shows a graphical overview of the FORMS elements and relations for the unification of the reflection and the distribution perspective.

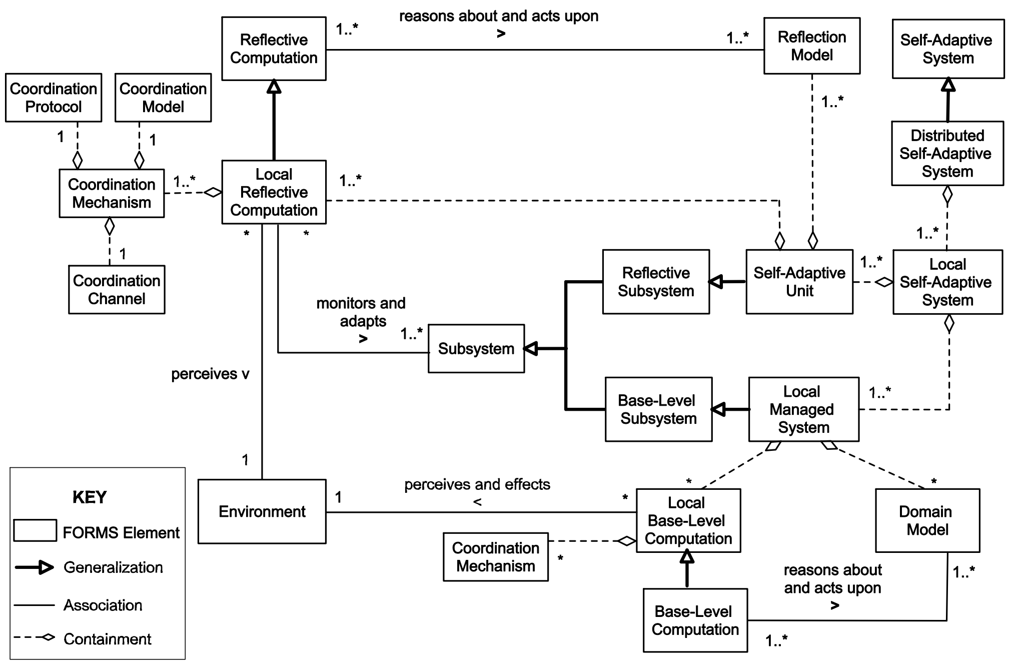


Fig. 7. FORMS: unification with distribution perspective.

B.1 Coordination Mechanism

We define a coordination mechanism as follows:

$$\begin{array}{|l}
\hline
CoordinationMechanism\ [Protocol, Model, Channel] \\
\hline
protocol : Protocol \\
model : Model \\
channel : Channel \\
\hline
\end{array}$$

A coordination mechanism comprises a coordination protocol, a coordination model, and a coordination channel.

B.2 Local Managed System

Local base-level computation extends base-level computation with a coordination mechanism, enablinmg it to exchange messages with other base-level computations:

$$\begin{array}{|l}
\hline
LocalBaseLevelComputation\ [Protocol, Model, Channel] \\
\hline
BaseLevelComputation \\
coordinationMechanism : CoordinationMechanism[Protocol, Model, Channel] \\
\hline
\end{array}$$

A local managed system is defined as:

$$
\begin{array}{|l}
\hline
\mathit{LocalManagedSystem}\ [\mathit{Protocol}, \mathit{Model}, \mathit{Channel}] \\
\mathit{models} : \mathbb{P}\ \mathit{DomainModel} \\
\mathit{computations} : \\
\qquad \mathbb{P}\ \mathit{LocalBaseLevelComputation}[\mathit{Protocol}, \mathit{Model}, \mathit{Channel}] \\
\hline
\forall\, c : \mathit{computations} \bullet \\
\qquad \mathrm{dom}\ c.\mathit{read} = \{\mathit{mdls} : \mathbb{P}\ \mathit{DomainModel} \mid \mathit{mdls} \subseteq \mathit{models} \bullet \\
\qquad\qquad (\mathit{mdls}, c.\mathit{state})\} \land \\
\qquad \mathrm{dom}\ c.\mathit{write} = \{\mathit{mdls} : \mathbb{P}\ \mathit{DomainModel} \mid \mathit{mdls} \subseteq \mathit{models} \bullet \\
\qquad\qquad (c.\mathit{state}, \mathit{mdls})\} \\
\hline
\end{array}
$$

A local managed system is a base-level subsystem comprising a set of domain models and a set of local base-level computations.

B.3 Self-Adaptive Unit

A local reflective computation is a reflective computation that comprises a coordination mechanism:

$$
\begin{array}{|l}
\hline
\mathit{LocalReflectiveComputation}\ [\mathit{Subsystem}, \mathit{Protocol}, \mathit{Model}, \mathit{Channel}] \\
\mathit{Computation} \\
\mathit{coordinationMechanism} : \\
\qquad \mathit{CoordinationMechanism}[\mathit{Protocol}, \mathit{Model}, \mathit{Channel}] \\
\mathit{read} : \mathbb{P}\ \mathit{ReflectionModel} \times \mathbb{P}\ \mathit{State} \to \mathbb{P}\ \mathit{State} \\
\mathit{write} : \mathbb{P}\ \mathit{State} \times \mathbb{P}\ \mathit{ReflectionModel} \to \mathbb{P}\ \mathit{ReflectionModel} \\
\mathit{perceive} : \mathit{Context} \times \mathbb{P}\ \mathit{State} \to \mathbb{P}\ \mathit{State} \\
\mathit{sense} : \mathbb{P}\ \mathit{Subsystem} \times \mathbb{P}\ \mathit{State} \to \mathbb{P}\ \mathit{State} \\
\mathit{adapt} : \mathbb{P}\ \mathit{Subsystem} \times \mathbb{P}\ \mathit{State} \to \mathbb{P}\ \mathit{Subsystem} \\
\mathit{trigger} : \mathbb{P}\ \mathit{State} \times \mathbb{P}\ \mathit{LocalReflectiveComputation}[\\
\qquad \mathit{Subsystem}, \mathit{Protocol}, \mathit{Model}, \mathit{Channel}] \to \\
\qquad\qquad \mathbb{P}\ \mathit{LocalReflectiveComputation}[\mathit{Subsystem}, \mathit{Protocol}, \mathit{Model}, \mathit{Channel}] \\
\hline
\end{array}
$$

The self-adaptive unit is defined as:

$$
\begin{array}{|l}
\hline
\mathit{SelfAdaptiveUnit}\ [\mathit{Subsystem}, \mathit{Protocol}, \mathit{Model}, \mathit{Channel}] \\
\mathit{models} : \mathbb{P}\ \mathit{ReflectionModel} \\
\mathit{computations} : \mathbb{P}\ \mathit{LocalReflectiveComputation}[\\
\qquad \mathit{Subsystem}, \mathit{Protocol}, \mathit{Model}, \mathit{Channel}] \\
\hline
\forall\, c : \mathit{computations} \bullet \\
\qquad \mathrm{dom}\ c.\mathit{read} = \{\mathit{mdls} : \mathbb{P}\ \mathit{ReflectionModel} \mid \mathit{mdls} \subseteq \mathit{models} \bullet \\
\qquad\qquad (\mathit{mdls}, c.\mathit{state})\} \land \\
\qquad \mathrm{dom}\ c.\mathit{write} = \{\mathit{mdls} : \mathbb{P}\ \mathit{ReflectionModel} \mid \mathit{mdls} \subseteq \mathit{models} \bullet \\
\qquad\qquad (c.\mathit{state}, \mathit{mdls})\} \land \\
\qquad \mathrm{dom}\ c.\mathit{trigger} = \{\mathit{ct} : \mathbb{P}\ \mathit{LocalReflectiveComputation}[\\
\qquad\qquad \mathit{Subsystem}, \mathit{Protocol}, \mathit{Model}, \mathit{Channel}] \mid \\
\qquad\qquad\qquad \mathit{ct} \subseteq \mathit{computations} \setminus \{c\} \bullet (c.\mathit{state}, \mathit{ct})\} \\
\hline
\end{array}
$$

A self-adaptive unit is a reflective subsystem comprising reflection models and local reflective computations.

B.4 Distributed Self-Adaptive System

A local self-adaptive systems comprises a set of local managed systems and a set of self-adaptive units. As an example, we consider a local self-adaptive system with one reflective layer in which all base level computations use a particular coordination mechanism and all reflective computations use a particular coordination protocol:

$$\begin{array}{l}
\mathit{LocalSelfAdaptiveSystem}\ [BCP, BCM, BCC, ACP, ACM, ACC] \\
\hline
\mathit{localManagedSystems} : \mathbb{P}\ \mathit{LocalManagedSystem}[BCP, BCM, BCC] \\
\mathit{selfAdaptiveUnits} : \\
\quad \mathbb{P}\ \mathit{SelfAdaptiveUnit}[\mathit{LocalManagedSystem}, ACP, ACM, ACC] \\
\hline
\forall\ \mathit{sau} : \mathit{selfAdaptiveUnits};\ \mathit{lrcs}, \mathit{lrca} : \mathit{LocalReflectiveComputation} \bullet \\
\quad \mathit{lrcs} \in \mathit{sau.computations} \wedge \mathit{lrca} \in \mathit{sau.computations} \wedge \\
\quad \mathrm{dom}\ \mathit{lrcs.sense} = \{\mathit{lms} : \mathbb{P}\ \mathit{LocalManagedSystem} \mid \\
\qquad \mathit{lms} \subseteq \mathit{localManagedSystems} \bullet (\mathit{lms}, \mathit{lrcs.state})\} \wedge \\
\quad \mathrm{dom}\ \mathit{lrca.adapt} = \{\mathit{lms} : \mathbb{P}\ \mathit{LocalManagedSystem} \mid \\
\qquad \mathit{lms} \subseteq \mathit{localManagedSystems} \bullet (\mathit{lms}, \mathit{lrca.state})\}
\end{array}$$

The abbreviations BCP, BCM, and BCC refer respectively to the coordination protocol, coordination model, and coordination channel for the base level system. ACP, ACM, and ACC are similar abbreviations for the coordination elements of the self-adaptive unit.

The specification states that self-adaptive units can sense and adapt the local managed systems of the local self-adaptive system.

A situated local self-adaptive system is a local self-adaptive system situated in some context of the environment:

$$\begin{array}{l}
\mathit{SitutatedLocalSelfAdaptiveSystem}\ [BCP, BCM, BCC, ACP, ACM, ACC] \\
\hline
\mathit{Environment} \\
\mathit{LocalSelfAdaptiveSystem}[BCP, BCM, BCC, ACP, ACM, ACC] \\
\mathit{context} : \mathit{Context} \\
\hline
\mathit{context} \subseteq \mathit{attributes} \\
\forall\ \mathit{lms} : \mathit{localManagedSystems};\ c : \mathit{LocalBaseLevelComputation} \bullet \\
\quad c \in \mathit{lms.computations} \wedge \\
\quad \mathrm{dom}\ c.\mathit{perceive} = \\
\qquad \{\mathit{attrs} : \mathit{Context} \mid \mathit{attrs} \subseteq \mathit{context} \bullet (c.\mathit{state}, \mathit{attrs})\} \wedge \\
\quad \mathrm{dom}\ c.\mathit{effect} = \\
\qquad \{\mathit{attrs} : \mathit{Context} \mid \mathit{attrs} \subseteq \mathit{context} \bullet (c.\mathit{state}, \mathit{attrs})\} \\
\forall\ \mathit{sau} : \mathit{selfAdaptiveUnits};\ \mathit{lrc} : \mathit{LocalReflectiveComputation} \bullet \\
\quad \mathit{lrc} \in \mathit{sau.computations} \wedge \\
\quad \mathrm{dom}\ \mathit{lrc.perceive} = \\
\qquad \{\mathit{attrs} : \mathit{Context} \mid \mathit{attrs} \subseteq \mathit{context} \bullet (\mathit{attrs}, \mathit{lrc.state})\}
\end{array}$$

A distributed self-adaptive system comprises a set of local self-adaptive systems:

$$\begin{array}{l}
\mathit{DistributedSelfAdaptiveSystem}\ [BCP, BCM, BCC, ACP, ACM, ACC] \\
\hline
\mathit{localSelfAdaptiveSystems} : \\
\quad \mathbb{P}\ \mathit{LocalSelfAdaptiveSystem}[BCP, BCM, BCC, ACP, ACM, ACC]
\end{array}$$

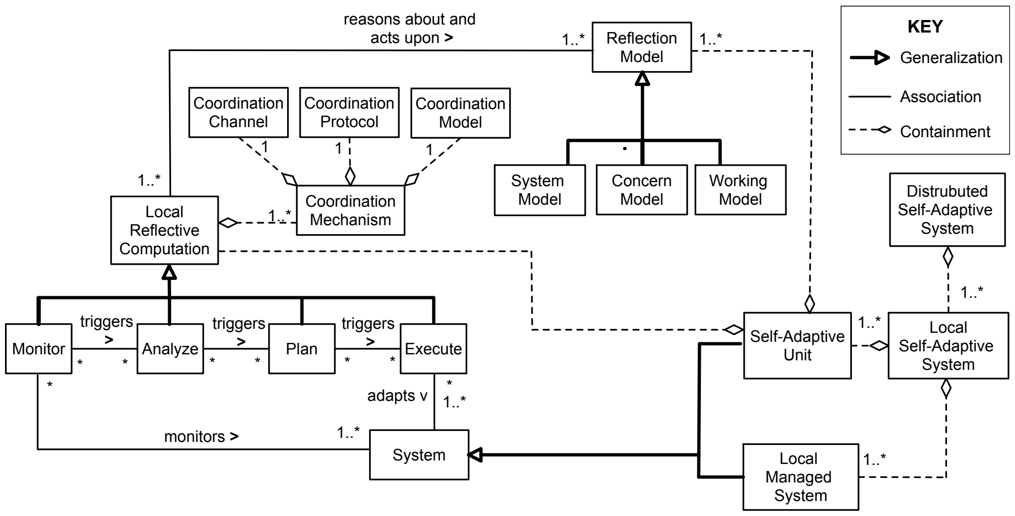


Fig. 8. FORMS: unification with MAPE perspective.

C. UNIFICATION WITH MAPE-K PERSPECTIVE

Fig. 8 shows a graphical overview of FORMS elements and relations integrated with the MAPE perspective.

C.1 Reflection Models

We distinguish between four types of reflection models: environment model, concern model, mape working model, and subsystem model. To describe reflection models, we first introduce a number of additional types of representations:

$[ConcernRepresentation, MapeRepresentation]$

A concern representation is a representation of a particular concern of interest. Mape representations are used to describe working models used by reflective computations.

A subsystem representation is a representation of (a part of) a subsystem which can be either a base-level subsystem or a reflective subsystem. Subsystem representations are defined:

$SubsystemRepresentation\ [Subsystem]$

A environment model comprises representations of attributes in the environment relevant for a particular concern of interest. Environment models are defined:

$EnvironmentModel$

$Environment$
$Model[EnvironmentRepresentation]$
$mapping : \mathbb{P}\, Attribute \leftrightarrow EnvironmentRepresentation$

$\operatorname{dom} mapping \subseteq \{attrs : \mathbb{P}\, Attribute \mid attrs \subseteq attributes\}$
$\operatorname{ran} mapping = \{r : EnvironmentRepresentation \mid r \in representations\}$

A concern model models a particular concern of interest. Concern models are defined:

$$\begin{array}{|l}\hline \textit{ConcernModel} \\ \textit{Model}[\textit{ConcernRepresentation}] \\ \hline\end{array}$$

A mape working model is a model used by reflective computations to deal with a concern of interest. Mape working models are defined:

$$\begin{array}{|l}\hline \textit{MapeWorkingModel} \\ \textit{Model}[\textit{MapeRepresentation}] \\ \hline\end{array}$$

To define a subsystem model we first introduce the concept of feature. Features describe perceivable characteristics of software systems:

$[\textit{Feature}]$

We define a function *reify* that returns the features for a given subsystem:

$$\begin{array}{|l}\hline [\textit{Subsystem}] \\ \textit{reify} : \textit{Subsystem} \rightarrow \mathbb{P}\,\textit{Feature}\end{array}$$

A subsystem model is a model of a subsystem (either a base-level system or a reflective subsystem). Subsystem models are defined:

$$\begin{array}{|l}\hline \textit{SubsystemModel}\ [\textit{Subsystem}] \\ \textit{subsystem} : \textit{Subsystem} \\ \textit{Model}[\textit{SubsystemRepresentation}[\textit{Subsystem}]] \\ \textit{mapping} : \mathbb{P}\,\textit{Feature} \leftrightarrow \textit{SubsystemRepresentation}[\textit{Subsystem}] \\ \hline \operatorname{dom} \textit{mapping} \subseteq \{\textit{features} : \mathbb{P}\,\textit{Feature} \mid \textit{features} \subseteq \textit{reify}(\textit{subsystem})\} \\ \operatorname{ran} \textit{mapping} = \{r : \textit{SubsystemRepresentation}[\textit{Subsystem}] \mid \\ \qquad r \in \textit{representations}\} \\ \hline\end{array}$$

A base-level subsystem model is defined:

$$\begin{array}{|l}\hline \textit{BaseLevelSubsystemModel} \\ \textit{SubsystemModel}[\textit{BaseLevelSubsystem}] \\ \hline\end{array}$$

We introduce reflection models which groups the sets of models used by a set of reflective computations:

$$\begin{array}{|l}\hline \textit{ReflectionModels}\ [\textit{Subsystem}] \\ \textit{environmentModels} : \mathbb{P}\,\textit{EnvironmentModel} \\ \textit{concernModels} : \mathbb{P}\,\textit{ConcernModel} \\ \textit{mapeWorkingModels} : \mathbb{P}\,\textit{MapeWorkingModel} \\ \textit{subsystemModels} : \mathbb{P}\,\textit{SubsystemModel}[\textit{Subsystem}] \\ \hline\end{array}$$

C.2 Reflective Computations

We define five types of reflective computations for self-adaptive system: update, monitor, analyse, plan and execute.

$$\begin{array}{l}
\textit{Update} \\
\hline
\textit{Computation} \\
\textit{read} : \mathbb{P}\,\textit{EnvironmentModel} \times \mathbb{P}\,\textit{State} \rightarrow \mathbb{P}\,\textit{State} \\
\textit{write} : \mathbb{P}\,\textit{State} \times \mathbb{P}\,\textit{EnvironmentModel} \rightarrow \mathbb{P}\,\textit{EnvironmentModel} \\
\textit{perceive} : \textit{Context} \times \mathbb{P}\,\textit{State} \rightarrow \mathbb{P}\,\textit{State} \\
\hline
\end{array}$$

Update computations perceive the environment and update the environment models accordingly.

$$\begin{array}{l}
\textit{Monitor}\ [\textit{Subsystem}] \\
\hline
\textit{Computation} \\
\textit{read} : \mathbb{P}\,\textit{MapeWorkingModel} \times \mathbb{P}\,\textit{SubsystemModel}[\textit{Subsystem}] \times \mathbb{P}\,\textit{State} \\
\qquad \rightarrow \mathbb{P}\,\textit{State} \\
\textit{write} : \mathbb{P}\,\textit{State} \times \mathbb{P}\,\textit{MapeWorkingModel} \times \mathbb{P}\,\textit{SubsystemModel}[\textit{Subsystem}] \\
\qquad \rightarrow \mathbb{P}\,\textit{MapeWorkingModel} \times \mathbb{P}\,\textit{SubsystemModel}[\textit{Subsystem}] \\
\textit{sense} : \mathbb{P}\,\textit{Subsystem} \times \mathbb{P}\,\textit{State} \rightarrow \mathbb{P}\,\textit{State} \\
\textit{trigger} : \mathbb{P}\,\textit{State} \times \mathbb{P}\,\textit{Analyse}[\textit{Subsystem}] \rightarrow \mathbb{P}\,\textit{Analyse}[\textit{Subsystem}] \\
\hline
\end{array}$$

Monitor computations monitor the underlying subsystem and maintain the subsystem models and possibly mape working models. Monitor computations can trigger analyse computations in particular states.

$$\begin{array}{l}
\textit{Analyse}\ [\textit{Subsystem}] \\
\hline
\textit{Computation} \\
\textit{read} : \mathbb{P}\,\textit{EnvironmentModel} \times \mathbb{P}\,\textit{ConcernModel} \times \mathbb{P}\,\textit{MapeWorkingModel} \times \\
\qquad \mathbb{P}\,\textit{SubsystemModel}[\textit{Subsystem}] \times \mathbb{P}\,\textit{State} \rightarrow \mathbb{P}\,\textit{State} \\
\textit{write} : \mathbb{P}\,\textit{State} \times \mathbb{P}\,\textit{MapeWorkingModel} \rightarrow \mathbb{P}\,\textit{MapeWorkingModel} \\
\textit{trigger} : \mathbb{P}\,\textit{State} \times \mathbb{P}\,\textit{Plan}[\textit{Subsystem}] \rightarrow \mathbb{P}\,\textit{Plan}[\textit{Subsystem}] \\
\hline
\end{array}$$

$$\begin{array}{l}
\textit{Plan}\ [\textit{Subsystem}] \\
\hline
\textit{Computation} \\
\textit{read} : \mathbb{P}\,\textit{EnvironmentModel} \times \mathbb{P}\,\textit{ConcernModel} \times \mathbb{P}\,\textit{MapeWorkingModel} \times \\
\qquad \mathbb{P}\,\textit{SubsystemModel}[\textit{Subsystem}] \times \mathbb{P}\,\textit{State} \rightarrow \mathbb{P}\,\textit{State} \\
\textit{write} : \mathbb{P}\,\textit{State} \times \mathbb{P}\,\textit{ConcernModel} \times \mathbb{P}\,\textit{MapeWorkingModel} \\
\qquad \rightarrow \mathbb{P}\,\textit{ConcernModel} \times \mathbb{P}\,\textit{MapeWorkingModel} \\
\textit{trigger} : \mathbb{P}\,\textit{State} \times \mathbb{P}\,\textit{Execute}[\textit{Subsystem}] \rightarrow \mathbb{P}\,\textit{Execute}[\textit{Subsystem}] \\
\hline
\end{array}$$

Analyse and plan computations reason about and act upon the reflection models in order to deal with the concerns of the self-adaptive system. Analyse computations can trigger plan computations in particular states, while plan computations can trigger execute computations.

$$
\begin{array}{l}
\textit{Execute}\ [\textit{Subsystem}] \\
\hline
\textit{Computation} \\
\textit{read} : \mathbb{P}\ \textit{EnvironmentModel} \times \mathbb{P}\ \textit{MapeWorkingModel} \times \\
\qquad \mathbb{P}\ \textit{SubsystemModel}[\textit{Subsystem}] \times \mathbb{P}\ \textit{State} \rightarrow \mathbb{P}\ \textit{State} \\
\textit{write} : \mathbb{P}\ \textit{State} \times \mathbb{P}\ \textit{MapeWorkingModel} \times \mathbb{P}\ \textit{SubsystemModel}[\textit{Subsystem}] \\
\qquad \rightarrow \mathbb{P}\ \textit{MapeWorkingModel} \times \mathbb{P}\ \textit{SubsystemModel}[\textit{Subsystem}] \\
\textit{adapt} : \mathbb{P}\ \textit{Subsystem} \times \mathbb{P}\ \textit{State} \rightarrow \mathbb{P}\ \textit{Subsystem} \\
\hline
\end{array}
$$

Execute computations use environment models and mape working models to adapt the underlying subsystem.

We define the sets of computations of a reflective subsystem for each type of reflective computation.

$$
\begin{array}{l}
\textit{Updating}\ [\textit{Subsystem}] \\
\hline
\textit{updates} : \mathbb{P}\ \textit{Update} \\
\textit{ReflectionModels}[\textit{Subsystem}] \\
\hline
\forall\, u : \textit{updates} \bullet \\
\qquad \mathrm{dom}\, u.\textit{read} = \{\textit{eModels} : \mathbb{P}\ \textit{EnvironmentModel} \mid \\
\qquad\qquad \textit{eModels} \subseteq \textit{environmentModels} \bullet (\textit{eModels}, u.\textit{state})\} \wedge \\
\qquad \mathrm{dom}\, u.\textit{write} = \{\textit{eModels} : \mathbb{P}\ \textit{EnvironmentModel} \mid \\
\qquad\qquad \textit{eModels} \subseteq \textit{environmentModels} \bullet (u.\textit{state}, \textit{eModels})\} \\
\hline
\end{array}
$$

Update computations act upon (a subset of) the environment models.

$$
\begin{array}{l}
\textit{Monitoring}\ [\textit{Subsystem}] \\
\hline
\textit{monitors} : \mathbb{P}\ \textit{Monitor} \\
\textit{ReflectionModels}[\textit{Subsystem}] \\
\hline
\forall\, m : \textit{monitors} \bullet \\
\qquad \mathrm{dom}\, m.\textit{read} = \{\textit{mModels} : \mathbb{P}\ \textit{MapeWorkingModel}; \\
\qquad\qquad \textit{sModels} : \mathbb{P}\ \textit{SubsystemModel}[\textit{Subsystem}] \mid \\
\qquad\qquad\qquad \textit{mModels} \subseteq \textit{mapeWorkingModels} \wedge \\
\qquad\qquad\qquad \textit{sModels} \subseteq \textit{subsystemModels} \bullet \\
\qquad\qquad\qquad\qquad (\textit{mModels}, \textit{sModels}, m.\textit{state})\} \wedge \\
\qquad \mathrm{dom}\, m.\textit{write} = \{\textit{mModels} : \mathbb{P}\ \textit{MapeWorkingModel}; \\
\qquad\qquad \textit{sModels} : \mathbb{P}\ \textit{SubsystemModel}[\textit{Subsystem}] \mid \\
\qquad\qquad\qquad \textit{mModels} \subseteq \textit{mapeWorkingModels} \wedge \\
\qquad\qquad\qquad \textit{sModels} \subseteq \textit{subsystemModels} \bullet \\
\qquad\qquad\qquad\qquad (m.\textit{state}, \textit{mModels}, \textit{sModels})\} \\
\hline
\end{array}
$$

Monitor computations act upon subsystem models and mape working models.

$$
\begin{array}{l}
\hline
\quad \mathit{Analyzing}\,[\mathit{Subsystem}] \\
\quad \mathit{analyses} : \mathbb{P}\, \mathit{Analyse} \\
\quad \mathit{ReflectionModels}[\mathit{Subsystem}] \\
\hline
\quad \forall\, a : \mathit{analyses} \bullet \\
\qquad\quad \mathrm{dom}\, a.\mathit{read} = \{ \mathit{eModels} : \mathbb{P}\, \mathit{EnvironmentModel}; \\
\qquad\qquad\quad \mathit{cModels} : \mathbb{P}\, \mathit{ConcernModel}; \\
\qquad\qquad\quad \mathit{mModels} : \mathbb{P}\, \mathit{MapeWorkingModel}; \\
\qquad\qquad\quad \mathit{sModels} : \mathbb{P}\, \mathit{SubsystemModel}[\mathit{Subsystem}] \mid \\
\qquad\qquad\qquad\quad \mathit{eModels} \subseteq \mathit{environmentModels} \land \\
\qquad\qquad\qquad\quad \mathit{cModels} \subseteq \mathit{concernModels} \land \\
\qquad\qquad\qquad\quad \mathit{mModels} \subseteq \mathit{mapeWorkingModels} \land \\
\qquad\qquad\qquad\quad \mathit{sModels} \subseteq \mathit{subsystemModels} \bullet \\
\qquad\qquad\qquad\qquad\quad (\mathit{eModels}, \mathit{cModels}, \mathit{mModels}, \mathit{sModels}, a.\mathit{state}) \} \land \\
\qquad\quad \mathrm{dom}\, a.\mathit{write} = \{ \mathit{mModels} : \mathbb{P}\, \mathit{MapeWorkingModel} \mid \\
\qquad\qquad\qquad\quad \mathit{mModels} \subseteq \mathit{mapeWorkingModels} \bullet (a.\mathit{state}, \mathit{mModels}) \} \\
\hline
\end{array}
$$

Analyse computations read the different kinds of reflection models and write its analysis results to the mape working models.

$$
\begin{array}{l}
\hline
\quad \mathit{Planning}\,[\mathit{Subsystem}] \\
\quad \mathit{plans} : \mathbb{P}\, \mathit{Plan} \\
\quad \mathit{ReflectionModels}[\mathit{Subsystem}] \\
\hline
\quad \forall\, p : \mathit{plans} \bullet \\
\qquad\quad \mathrm{dom}\, p.\mathit{read} = \{ \mathit{eModels} : \mathbb{P}\, \mathit{EnvironmentModel}; \\
\qquad\qquad\quad \mathit{cModels} : \mathbb{P}\, \mathit{ConcernModel}; \\
\qquad\qquad\quad \mathit{mModels} : \mathbb{P}\, \mathit{MapeWorkingModel}; \\
\qquad\qquad\quad \mathit{sModels} : \mathbb{P}\, \mathit{SubsystemModel}[\mathit{Subsystem}] \mid \\
\qquad\qquad\qquad\quad \mathit{eModels} \subseteq \mathit{environmentModels} \land \\
\qquad\qquad\qquad\quad \mathit{cModels} \subseteq \mathit{concernModels} \land \\
\qquad\qquad\qquad\quad \mathit{mModels} \subseteq \mathit{mapeWorkingModels} \land \\
\qquad\qquad\qquad\quad \mathit{sModels} \subseteq \mathit{subsystemModels} \bullet \\
\qquad\qquad\qquad\qquad\quad (\mathit{eModels}, \mathit{cModels}, \mathit{mModels}, \mathit{sModels}, p.\mathit{state}) \} \land \\
\qquad\quad \mathrm{dom}\, p.\mathit{write} = \{ \mathit{cModels} : \mathbb{P}\, \mathit{ConcernModel}; \\
\qquad\qquad\quad \mathit{mModels} : \mathbb{P}\, \mathit{MapeWorkingModel} \mid \\
\qquad\qquad\qquad\quad \mathit{cModels} \subseteq \mathit{concernModels} \land \\
\qquad\qquad\qquad\quad \mathit{mModels} \subseteq \mathit{mapeWorkingModels} \bullet \\
\qquad\qquad\qquad\qquad\quad (p.\mathit{state}, \mathit{cModels}, \mathit{mModels}) \} \\
\hline
\end{array}
$$

Plan computations use the different reflection models to update the concern models and mape working models.

$$
\begin{array}{l}
\textit{Executing}\ [\textit{Subsystem}] \\
\hline
\textit{executes} : \mathbb{P}\, \textit{Execute} \\
\textit{ReflectionModels}[\textit{Subsystem}] \\
\hline
\forall\, e : \textit{executes} \bullet \\
\quad \mathrm{dom}\, e.\textit{read} = \{\textit{eModels} : \mathbb{P}\, \textit{EnvironmentModel}; \\
\qquad \textit{mModels} : \mathbb{P}\, \textit{MapeWorkingModel}; \\
\qquad \textit{sModels} : \mathbb{P}\, \textit{SubsystemModel}[\textit{Subsystem}] \mid \\
\qquad\quad \textit{eModels} \subseteq \textit{environmentModels} \land \\
\qquad\quad \textit{mModels} \subseteq \textit{mapeWorkingModels} \land \\
\qquad\quad \textit{sModels} \subseteq \textit{subsystemModels} \bullet \\
\qquad\qquad (\textit{eModels}, \textit{mModels}, \textit{sModels}, e.\textit{state})\} \land \\
\quad \mathrm{dom}\, e.\textit{write} = \{\textit{mModels} : \mathbb{P}\, \textit{MapeWorkingModel}; \\
\qquad \textit{sModels} : \mathbb{P}\, \textit{SubsystemModel}[\textit{Subsystem}] \mid \\
\qquad\quad \textit{mModels} \subseteq \textit{mapeWorkingModels} \land \\
\qquad\quad \textit{sModels} \subseteq \textit{subsystemModels} \bullet \\
\qquad\qquad (e.\textit{state}, \textit{mModels}, \textit{sModels})\}
\end{array}
$$

To perform adaptations, execute computations use the information of the different reflection models. An execute computation can maintain a subsystem model while performing adaptations of the corresponding subsystem.

The reflective computations schema groups the computations of a reflective subsystem:

$$
\begin{array}{l}
\textit{ReflectiveComputations}\ [\textit{Subsystem}] \\
\hline
\textit{Updating}[\textit{Subsystem}] \\
\textit{Monitoring}[\textit{Subsystem}] \\
\textit{Analyzing}[\textit{Subsystem}] \\
\textit{Planning}[\textit{Subsystem}] \\
\textit{Executing}[\textit{Subsystem}] \\
\hline
\forall\, m : \textit{monitors} \bullet \\
\quad \mathrm{dom}\, m.\textit{trigger} = \{\textit{as} : \mathbb{P}\, \textit{Analyse} \mid \textit{as} \subseteq \textit{analyses} \bullet (m.\textit{state}, \textit{as})\} \\
\forall\, a : \textit{analyses} \bullet \\
\quad \mathrm{dom}\, a.\textit{trigger} = \{\textit{ps} : \mathbb{P}\, \textit{Plan} \mid \textit{ps} \subseteq \textit{plans} \bullet (a.\textit{state}, \textit{ps})\} \\
\forall\, p : \textit{plans} \bullet \\
\quad \mathrm{dom}\, p.\textit{trigger} = \{\textit{es} : \mathbb{P}\, \textit{Execute} \mid \textit{es} \subseteq \textit{executes} \bullet (p.\textit{state}, \textit{es})\}
\end{array}
$$

Triggers are restricted to (the subsets of) the respective computations of a reflective subsystem.

C.3 IBM's Autonomic Manager Framework

To conclude the MAPE-perspective, we formally describe an example of a hierarchical self-adaptive autonomic system.

The base-level subsystem in an autonomic self-adaptive system is a managed resource and is defined as:

$$
\begin{array}{l}
\textit{ManagedResource} \\
\hline
\textit{BaseLevelSubsystem}
\end{array}
$$

Knowledge is defined as:

$Knowledge$

$ReflectionModel$

An autonomic manager is abstractly defined as:

$AutonomicManager$

$knowledge : Knowledge$

Autonomic manager computation manages a managed element (i.e. either a managed resource or an autonomic manager) and is defined as:

$AutonomicManagerComputation\ [ManagedElement]$

$ReflectiveComputation[ManagedElement]$

We distinguish between two types of autonomic managers: orchestrating autonomic manager and resource manager, defined as follows:

$OrchestratingAutonomicManager$

$AutonomicManager$
$mapeComputations :$
$\quad \mathbb{P}\ AutonomicManagerComputation[AutonomicManager]$

$ResourseAutonomicManager$

$AutonomicManager$
$mapeComputations :$
$\quad \mathbb{P}\ AutonomicManagerComputation[ManagedResource]$
$manage : Knowledge \times \mathbb{P}\ ManagedResource \rightarrow \mathbb{P}\ ManagedResource$

IBM's autonomic manager framework considers four different types of resource managers that deal with different types of concerns: self-healing, self-optimizing, self-healing, and self-protecting, These managers are defined as:

$SelfConfiguringAutonomicManager$

$ResourseAutonomicManager$

$SelfOptimizingAutonomicManager$

$ResourseAutonomicManager$

$SelfHealingAutonomicManager$

$ResourseAutonomicManager$

$SelfProtectingAutonomicManager$

$ResourseAutonomicManager$

For the example, we define a concrete type of orchestrating autonomic managers that manage resource autonomic manager for a single concern:

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{SingleConcernAutonomicManager}} \\
\mathit{OrchestratingAutonomicManager} \\
\mathit{manage} : \mathit{Knowledge} \times \mathbb{P}\, \mathit{ResourseAutonomicManager} \\
\qquad\qquad \rightarrow \mathbb{P}\, \mathit{ResourseAutonomicManager} \\
\hline
\end{array}
$$

Finally, we can specify a concrete self-adaptive autonomic system:

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{SelfAdaptiveAutonomicSystem}} \\
\mathit{Environment} \\
\mathit{context} : \mathit{Context} \\
\mathit{resources} : \mathbb{P}\, \mathit{ManagedResource} \\
\mathit{endpointManagers} : \mathbb{P}\, \mathit{ResourseAutonomicManager} \\
\mathit{systemManager} : \mathbb{P}\, \mathit{SingleConcernAutonomicManager} \\
\mathit{server}, \mathit{client}1, \mathit{client}2, \mathit{network} : \mathit{ManagedResource} \\
\mathit{serverOptimizer}, \mathit{networkOptimizer} : \mathit{ResourseAutonomicManager} \\
\mathit{systemOptimizer} : \mathit{SingleConcernAutonomicManager} \\
\hline
\mathit{resources} = \{\mathit{server}, \mathit{client}1, \mathit{client}2, \mathit{network}\} \\
\mathit{endpointManagers} = \{\mathit{serverOptimizer}, \mathit{networkOptimizer}\} \\
\mathit{systemManager} = \{\mathit{systemOptimizer}\} \\
\mathrm{dom}\, \mathit{serverOptimizer}.\mathit{manage} = \{(\mathit{serverOptimizer}.\mathit{knowledge}, \{\mathit{server}\})\} \\
\mathrm{ran}\, \mathit{serverOptimizer}.\mathit{manage} = \{\{\mathit{server}\}\} \\
\mathrm{dom}\, \mathit{networkOptimizer}.\mathit{manage} = \{(\mathit{networkOptimizer}.\mathit{knowledge}, \{\mathit{network}\})\} \\
\mathrm{ran}\, \mathit{networkOptimizer}.\mathit{manage} = \{\{\mathit{network}\}\} \\
\mathrm{dom}\, \mathit{systemOptimizer}.\mathit{manage} = \\
\qquad \{(\mathit{systemOptimizer}.\mathit{knowledge}, \{\mathit{serverOptimizer}, \mathit{networkOptimizer}\})\} \\
\mathrm{ran}\, \mathit{systemOptimizer}.\mathit{manage} = \{\{\mathit{serverOptimizer}, \mathit{networkOptimizer}\}\} \\
\hline
\end{array}
$$

In this example, one resource manager is managing a server, another one is managing a network. In addition, there is the system manager who serves as an orchestrating autonomic manager, managing the two resource managers. The specification describes a hierarchy of autonomic managers and specifies the scope of adaptations of the execute computations (i.e. manage) of the autonomic managers in the self-adaptive autonomic system.

D. TRAFFIC MONITORING CASE STUDY

Figure 5 shows the FORMS model of the traffic monitoring case study. By extending the FORMS abstractions, we can precisely define the elements required to support self-healing.

Subsequently, we specify the elements of the traffic environment, the local traffic monitoring system that instantiates a local base-level subsystem, the self-healing manager that instantiates a local reflective computation, and the integrated traffic monitoring system. Then we declaratively specify how one of the cameras is healed after the failure of a neighboring camera.

For brevity, we limit the specification to the essence of what is needed to specify the self-healing scenario. For a complete specification of the FORMS model of the traffic monitoring system and the scenario, we refer the interested reader to [Weyns et al. 2010b].

D.1 Traffic Environment

We define the following attributes of the traffic environment:

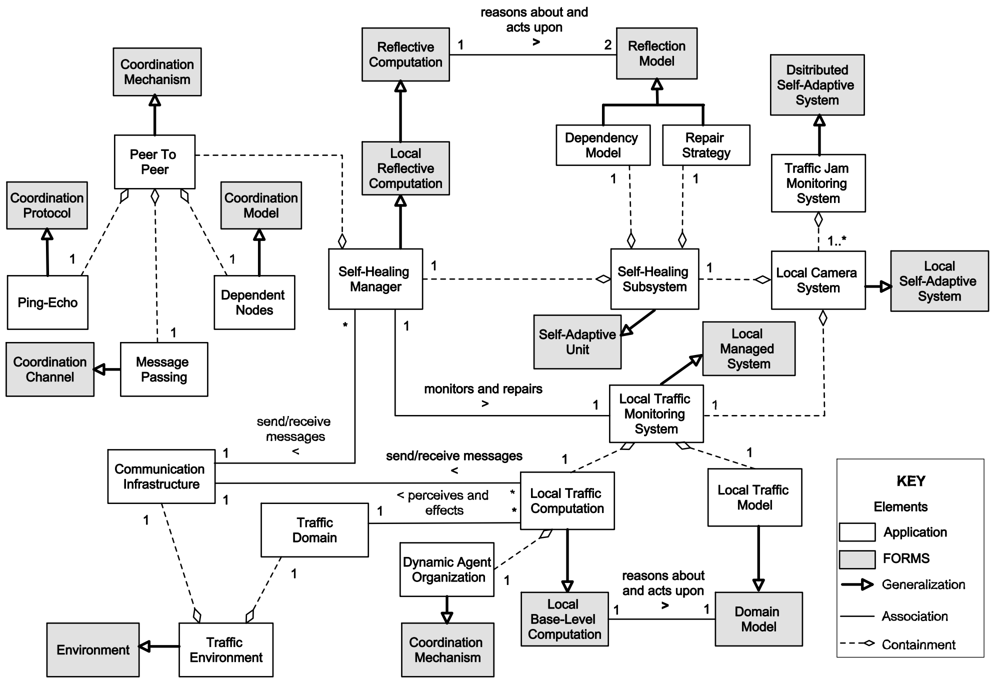


Fig. 9. FORMS model of the traffic - monitoring case.

$camera_1$, $camera_2$, $camera_3$, $freeflow\_zone_1$, $congested\_zone_1$, $freeflow\_zone_2$, $congested\_zone_2$, $freeflow\_zone_3$, $congested\_zone_3$, $congested\_zone_4$, $ping\_message_{12}$, $echo\_message_{21}$: $Attribute$

For brevity, we only define the attributes that we use further in the document. We introduce a name to group the attributes:

$traffic\_domain\_attributes == \{camera_1 , camera_2 , camera_3 , freeflow\_zone_1 , congested\_zone_1 , freeflow\_zone_2 , congested\_zone_2 , freeflow\_zone_3 , congested\_zone_3\}$

We consider the following traffic processes:

$traffic_1$, $traffic_2$, $traffic_3$, $monitor\_camera_1$, $monitor\_camera_2$, $monitor\_camera_3$, $transmit$: $Process$

$traffic\_domain\_processes == \{traffic_1 , traffic_2 , traffic_3 , monitor\_camera_1 , monitor\_camera_2 , monitor\_camera_3 , transmit\}$

Traffic processes represent the ongoing traffic in different monitored zones of the highway. A monitor camera process allows the observation of the traffic conditions in the viewing range of a camera. The transmit process provides the distributed communication service to transmit messages between cameras. This process is used by the local traffic monitoring systems to coordinate the agent organizations, and by the self-healing managers to coordinate for failure management.

A traffic environment is defined as an environment with traffic attributes and traffic processes:

$$\begin{array}{|l}
\textit{TrafficEnvironment} \\
\hline
\textit{Environment} \\
\hline
\textit{attributes} \subseteq \textit{traffic\_domain\_attributes} \\
\textit{processes} \subseteq \textit{traffic\_domain\_processes} \\
\hline
\end{array}$$

The traffic environment at T0 in the example (Figure 0**??**) is defined as:

$$\begin{array}{|l}
\textit{TrafficEnvironment}_{T0} \\
\hline
\textit{TrafficEnvironment} \\
\hline
\textit{attributes} = \{\textit{camera}_1\ , \textit{camera}_2\ , \textit{camera}_3\ , \textit{freeflow\_zone}_1\ , \textit{freeflow\_zone}_2\ , \\
\qquad \textit{congested\_zone}_3\} \\
\textit{processes} = \textit{traffic\_domain\_processes} \\
\hline
\end{array}$$

We introduce the type shutdown to model terminations of processes in the traffic environment:

$$\textit{Shutdown} : \mathbb{P}\ \textit{Process}$$

Similarly, we introduce the type startup to model the initiation of new processes in the environment:

$$\textit{Startup} : \mathbb{P}\ \textit{Process}$$

We consider one shutdown event in the traffic environment:

$$\begin{array}{|l}
\textit{shutdowns} : \mathbb{P}\ \textit{Shutdown} \\
\hline
\textit{shutdowns} = \{\textit{monitor\_camera}_2\}
\end{array}$$

We consider the following set of events in the traffic environment:

$$\begin{array}{|l}
\textit{events} : \mathbb{P}\ \textit{Event} \\
\hline
\textit{events} = \{\textit{traffic}_2 \mapsto (\{\textit{freeflow\_zone}_2\} \mapsto \{\textit{congested\_zone}_2\})\ , \\
\qquad\qquad \textit{monitor\_camera}_2 \mapsto (\{\textit{camera}_2\} \mapsto \{\})\}
\end{array}$$

The change of the traffic state in zone 2 at T1 is defined as:

$$\begin{array}{|l}
\textit{TrafficEnvironment}_{T1} \\
\hline
\Delta\, \textit{TrafficEnvironment}_{T0} \\
p? : \textit{Process} \\
c? : \textit{Change} \\
\hline
p? = \textit{traffic}_2 \\
c? = \{\textit{freeflow\_zone}_2\} \mapsto \{\textit{congested\_zone}_2\} \\
(p?, c?) \in \textit{events} \\
\textit{attributes}' = \textit{attributes} \setminus \textit{first}(c?) \cup \textit{second}(c?) \\
\textit{processes}' = \textit{processes} \\
\hline
\end{array}$$

The specification states that the traffic state is changed from free-flow to congested by the traffic process in zone 2 (i.e. the zone monitored by camera 2).

From T1 to T2, the traffic environment does not change:

$$\begin{array}{|l}
\textit{TrafficEnvironment}_{T2} \\
\hline
\Xi\, \textit{TrafficEnvironment}_{T1} \\
\hline
\end{array}$$

A failure of camera 2 changes the traffic environment as follows:

$$
\begin{array}{|l}
\textit{TrafficEnvironment}_{T3} \\
\hline
\Delta\, \textit{TrafficEnvironment}_{T2} \\
p? : \textit{Process} \\
c? : \textit{Change} \\
s? : \textit{shutdowns} \\
\hline
p? = \textit{monitor\_camera}_2 \\
c? = \{\textit{camera}_2\} \mapsto \{\} \\
(p?, c?) \in \textit{events} \\
s? = \textit{monitor\_camera}_2 \\
\textit{attributes}' = \textit{attributes} \setminus \textit{first}(c?) \cup \textit{second}(c?) \\
\textit{processes}' = \textit{processes} \setminus \{s?\} \\
\hline
\end{array}
$$

The specification states that after the event, camera 2 is no longer available for traffic monitoring, and consequently, the traffic monitoring process of camera 2 is shutdown.

D.2 Local Traffic Monitoring System

We consider the following traffic environment representations:

$$
\begin{array}{|l}
\textit{cam}_1 , \textit{cam}_2 , \textit{cam}_3 , \textit{fflow\_zone}_1 , \textit{congst\_zone}_1 , \textit{fflow\_zone}_2, \textit{congst\_zone}_2 , \\
\textit{fflow\_zone}_3 , \textit{congst\_zone}_3 , \textit{ping}_{12} , \textit{echo}_{21} : \textit{EnvironmentRepresentation}
\end{array}
$$

$$
\begin{array}{l}
\textit{traffic\_environment\_representations} == \{\textit{cam}_1 , \textit{cam}_2 , \textit{cam}_3 , \textit{fflow\_zone}_1 , \textit{congst\_zone}_1 , \\
\quad \textit{fflow\_zone}_2 , \textit{congst\_zone}_2 , \textit{fflow\_zone}_3 , \textit{congst\_zone}_3 , \textit{ping}_{12} , \textit{echo}_{21}\}
\end{array}
$$

Attribute sets and environment representations in the traffic monitoring case are mapped as follows:

$$
\begin{array}{l}
\textit{traffic\_attribute\_representation\_mapping} == \\
\quad \{\{\textit{camera}_1\} \mapsto \textit{cam}_1 , \{\textit{camera}_2\} \mapsto \textit{cam}_2 , \{\textit{camera}_3\} \mapsto \textit{cam}_3 , \\
\quad \{\textit{freeflow\_zone}_1\} \mapsto \textit{fflow\_zone}_1 , \{\textit{congested\_zone}_1\} \mapsto \textit{congst\_zone}_1 , \\
\quad \{\textit{freeflow\_zone}_2\} \mapsto \textit{fflow\_zone}_2 , \{\textit{congested\_zone}_2\} \mapsto \textit{congst\_zone}_2 , \\
\quad \{\textit{freeflow\_zone}_3\} \mapsto \textit{fflow\_zone}_3 , \{\textit{congested\_zone}_3\} \mapsto \textit{congst\_zone}_3\}
\end{array}
$$

A local traffic model is defined as follows:

$$
\begin{array}{|l}
\textit{LocalTrafficModel} \\
\hline
\textit{TrafficEnvironment} \\
\textit{Model}[\textit{EnvironmentRepresentation}] \\
\textit{mapping} : \mathbb{P}\, \textit{Attribute} \leftrightarrow \textit{EnvironmentRepresentation} \\
\hline
\textit{representations} \subseteq \textit{traffic\_environment\_representations} \\
\mathrm{dom}\, \textit{mapping} \subseteq \{\textit{attrs} : \mathbb{P}\, \textit{Attribute} \mid \textit{attrs} \subseteq \textit{attributes}\} \\
\mathrm{ran}\, \textit{mapping} = \{r : \textit{EnvironmentRepresentation} \mid r \in \textit{representations}\} \\
\hline
\end{array}
$$

Local traffic model represents attributes of the traffic environment and maps the attributes to traffic environment representations.

The local traffic model of camera 1 at time T2 is defined:

$$
\begin{array}{|l}
\hline
\mathit{LocalTrafficModelOne}_{T2} \\
\mathit{TrafficEnvironment}_{T2} \\
\mathit{Model}[\mathit{EnvironmentRepresentation}] \\
\mathit{mapping} : \mathbb{P}\,\mathit{Attribute} \leftrightarrow \mathit{EnvironmentRepresentation} \\
\hline
\mathit{representations} = \{\mathit{fflow\_zone}_1\ , \mathit{cam}_2, \mathit{cam}_3\} \land \\
\mathit{mapping} = \{\{\mathit{freeflow\_zone}_1\} \mapsto \mathit{fflow\_zone}_1\ , \\
\qquad \{\mathit{camera}_2\} \mapsto \mathit{cam}_2, \{\mathit{camera}_3\} \mapsto \mathit{cam}_3\} \\
\hline
\end{array}
$$

The local traffic model for camera 2 at time T1 is defined:

$$
\begin{array}{|l}
\hline
\mathit{LocalTrafficModelTwo}_{T2} \\
\mathit{TrafficEnvironment}_{T2} \\
\mathit{Model}[\mathit{EnvironmentRepresentation}] \\
\mathit{mapping} : \mathbb{P}\,\mathit{Attribute} \leftrightarrow \mathit{EnvironmentRepresentation} \\
\hline
\mathit{representations} = \{\mathit{congst\_zone}_2\ , \mathit{cam}_1\ , \mathit{cam}_3\} \land \\
\mathit{mapping} = \{\{\mathit{congested\_zone}_2\} \mapsto \mathit{congst\_zone}_2\ , \\
\qquad \{\mathit{camera}_1\} \mapsto \mathit{cam}_1\ , \{\mathit{camera}_3\} \mapsto \mathit{cam}_3\} \\
\hline
\end{array}
$$

And for camera 3:

$$
\begin{array}{|l}
\hline
\mathit{LocalTrafficModelThree}_{T2} \\
\mathit{TrafficEnvironment}_{T2} \\
\mathit{Model}[\mathit{EnvironmentRepresentation}] \\
\mathit{mapping} : \mathbb{P}\,\mathit{Attribute} \leftrightarrow \mathit{EnvironmentRepresentation} \\
\hline
\mathit{representations} = \{\mathit{congst\_zone}_2\ , \mathit{congst\_zone}_3\ , \mathit{cam}_1\ , \mathit{cam}_2\} \land \\
\mathit{mapping} = \{\{\mathit{congested\_zone}_2\} \mapsto \mathit{congst\_zone}_2\ , \\
\qquad \{\mathit{congested\_zone}_3\} \mapsto \mathit{congst\_zone}_3\ , \{\mathit{camera}_1\} \mapsto \mathit{cam}_1\ , \\
\qquad \{\mathit{camera}_2\} \mapsto \mathit{cam}_2\} \\
\hline
\end{array}
$$

To define local traffic computations, we first introduce abstract types for the coordinating elements used by the computations. Local traffic computations can play two types of roles:

$\mathit{Role} ::= \mathit{master} \mid \mathit{slave}$

The protocol used for coordination by the local traffic computations is defined:

$$
\begin{array}{|l}
\hline
\mathit{MasterSlave} \\
\mathit{role} : \mathit{Role} \\
\hline
\end{array}
$$

To define the coordination model, we introduce a simple type of names:

$\mathit{Name} == \mathbb{N}$

As we will define below, the names are associated with local camera systems. For brevity in the explanation, sometimes we associate names with cameras.

The model used for coordination by the local traffic computations is defined:

$$
\begin{array}{|l}
\hline
\mathit{OrganizationPartners} \\
\mathit{partners} : \mathbb{P}\,\mathit{Name} \\
\mathit{neighborOrganizations} : \mathbb{P}\,\mathit{Name} \\
\hline
\end{array}
$$

Partners are the names of members in the organization in which a camera currently is involved. A slave has only one partner, i.e. its master. The partners of a master are its slaves. Neighbor organizations are the names of the masters of the organizations at neighboring nodes. Only masters maintain references to their neighbor organizations. For slaves, the set of neighbor organizations is empty.

The coordination channel uses for coordination by the local traffic computations is defined:

$MessagePassing$

$links : Name \leftrightarrow EnvironmentRepresentation$

A communication link maps a name of a camera to its representation. In the example, we consider three concrete communication channels:

$$traffic\_communication\_channel == \{1 \mapsto cam_1\ , 2 \mapsto cam_2\ , 3 \mapsto cam_3\}$$

To define messages, we first introduce an abstract type to represent the content of messages:

$[Content]$

Messages are defined as:

$Message$

$from : Name$

$to : \mathbb{P}\, Name$

$content : Content$

A message contains the name of the sender, the names of the addressees, and a content.

With the above specified types we can define the coordination mechanism that is used by local traffic computations:

$DynamicAgentOrganizations$

$orgProtocol : MasterSlave$

$orgModel : OrganizationPartners$

$channel : MessagePassing$

$\forall\, p : orgModel.partners \bullet \exists\, l : channel.links \bullet first(l) = p\ \wedge$

$\forall\, norg : orgModel.neighborOrganizations \bullet \exists\, l : channel.links \bullet first(l) = norg$

The predicate states that there is a communication link with every partner in the organization, and for the masters, with the masters of neighbor organizations.

The organization of camera 1 at T2 is defined:

$DynamicAgentOrganizationOne_{T2}$

$DynamicAgentOrganizations$

$orgProtocol.role = master$

$orgModel.partners = \varnothing$

$orgModel.neighborOrganizations = \{3\}$

$channel.links = traffic\_communication\_channel \setminus \{1 \mapsto cam_1\}$

At T2, camera 1 is the master of a single member organization (see Figure 0**??**). The master of the neighbor organization is camera 3. Camera 1 has communication channels with the two other cameras in in the traffic monitoring system.

The other organizations are defined as follows:

$$\begin{array}{|l}
\hline
\mathit{DynamicAgentOrganizationTwo}_{T2} \\
\mathit{DynamicAgentOrganizations} \\
\hline
\mathit{orgProtocol.role} = \mathit{slave} \\
\mathit{orgModel.partners} = \{3\} \\
\mathit{orgModel.neighborOrganizations} = \varnothing \\
\mathit{channel.links} = \mathit{traffic\_communication\_channel} \setminus \{2 \mapsto \mathit{cam}_2\} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\mathit{DynamicAgentOrganizationThree}_{T2} \\
\mathit{DynamicAgentOrganizations} \\
\hline
\mathit{orgProtocol.role} = \mathit{master} \\
\mathit{orgModel.partners} = \{2\} \\
\mathit{orgModel.neighborOrganizations} = \{1\} \\
\mathit{channel.links} = \mathit{traffic\_communication\_channel} \setminus \{3 \mapsto \mathit{cam}_3\} \\
\hline
\end{array}$$

Finally, a local traffic computation is defined as:

$$\begin{array}{|l}
\hline
\mathit{LocalTrafficComputation} \\
\mathit{Computation} \\
\mathit{read} : \mathit{LocalTrafficModel} \times \mathbb{P}\,\mathit{State} \rightarrow \mathbb{P}\,\mathit{State} \\
\mathit{write} : \mathbb{P}\,\mathit{State} \times \mathit{LocalTrafficModel} \rightarrow \mathit{LocalTrafficModel} \\
\mathit{perceive} : \mathbb{P}\,\mathit{State} \times \mathit{Context} \rightarrow \mathbb{P}\,\mathit{State} \\
\mathit{effect} : \mathbb{P}\,\mathit{State} \times \mathit{Context} \rightarrow \mathit{Context} \\
\mathit{trafficCoordinationMechanism} : \mathit{DynamicAgentOrganizations} \\
\mathit{send} : \mathbb{P}\,\mathit{State} \rightarrow \mathit{Message} \\
\mathit{receive} : \mathit{Message} \rightarrow \mathbb{P}\,\mathit{State} \\
\hline
\end{array}$$

A local traffic computation can act upon a local traffic model. It can perceive and effect the context in which the camera is situated. Local traffic computations use dynamic agent organizations as a coordination mechanism to detect traffic jams in continues monitored zones. Coordination is done by means of exchanging messages.

The local traffic computation of camera 1 at T2 is defined:

$$\begin{array}{|l}
\hline
\mathit{LocalTrafficComputationOne}_{T2} \\
\mathit{LocalTrafficComputation} \\
\mathit{DynamicAgentOrganizationOne}_{T2} \\
\hline
\end{array}$$

For the other cameras:

$$\begin{array}{|l}
\hline
\mathit{LocalTrafficComputationTwo}_{T2} \\
\mathit{LocalTrafficComputation} \\
\mathit{DynamicAgentOrganizationTwo}_{T2} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\mathit{LocalTrafficComputationThree}_{T2} \\
\mathit{LocalTrafficComputation} \\
\mathit{DynamicAgentOrganizationThree}_{T2} \\
\hline
\end{array}$$

Using local traffic model and local traffic computations, we can now define local traffic monitoring system:

$$\begin{array}{|l}
\textit{LocalTrafficMonitoringSystem} \\
\hline
\textit{trafficModel} : \textit{LocalTrafficModel} \\
\textit{computation} : \textit{LocalTrafficComputation} \\
\hline
\operatorname{dom} \textit{computation.read} = \{(\textit{trafficModel}, \textit{computation.state})\} \land \\
\operatorname{dom} \textit{computation.write} = \{(\textit{computation.state}, \textit{trafficModel})\} \land \\
\operatorname{dom} \textit{computation.send} = \{\textit{computation.state}\} \\
\hline
\end{array}$$

The predicate states that a local traffic computation is restricted to act upon the local traffic model, and messages for coordination are produced based on the current state of the computation.

The local traffic monitoring systems at T2 are:

$$\begin{array}{|l}
\textit{LocalTrafficMonitoringSystemOne}_{T2} \\
\hline
\textit{LocalTrafficMonitoringSystem} \\
\textit{LocalTrafficModelOne}_{T2} \\
\textit{LocalTrafficComputationOne}_{T2} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{LocalTrafficMonitoringSystemTwo}_{T2} \\
\hline
\textit{LocalTrafficMonitoringSystem} \\
\textit{LocalTrafficModelTwo}_{T2} \\
\textit{LocalTrafficComputationTwo}_{T2} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{LocalTrafficMonitoringSystemThree}_{T2} \\
\hline
\textit{LocalTrafficMonitoringSystem} \\
\textit{LocalTrafficModelThree}_{T2} \\
\textit{LocalTrafficComputationThree}_{T2} \\
\hline
\end{array}$$

D.3 Self-Healing Subsystem

We define two types of reflection models in the traffic monitoring case: dependency model and repair strategy.

To define a dependency model, we introduce the dependency type:

$$\textit{Dependency} ::= \textit{neighbor} \mid \textit{neighbormaster} \mid \textit{mymaster} \mid \textit{myslave}$$

For the example, we limit the dependencies to neighboring nodes, masters of neighboring organizations (only for masters), and master-slave dependencies.

A dependency model maps dependencies to names of cameras and is defined:

$$\begin{array}{|l}
\textit{DependencyModel} \\
\hline
\textit{dependencies} : \textit{Dependency} \leftrightarrow \textit{Name} \\
\hline
\end{array}$$

The dependency model for camera 1 at T2 is defined:

$\mathit{DependencyModelOne}_{T2}$
$\mathit{DependencyModel}$

$$\mathit{dependencies} = \{\mathit{neighbor} \mapsto 2, \mathit{neighbormaster} \mapsto 3, \mathit{myslave} \mapsto 0, \mathit{mymaster} \mapsto 0\}$$

Camera 2 has a dependency with camera 2 as neighbor and with camera 3 as neighbor master of another organization. We use "0" to indicate that camera 2 currently has no dependencies with slaves or a master.

The dependency models for the other cameras are defined as:

$\mathit{DependencyModelTwo}_{T2}$
$\mathit{DependencyModel}$

$$\mathit{dependencies} = \{\mathit{neighbor} \mapsto 1, \mathit{neighbor} \mapsto 3, \mathit{myslave} \mapsto 0, \mathit{mymaster} \mapsto 3\}$$

$\mathit{DependencyModelThree}_{T2}$
$\mathit{DependencyModel}$

$$\mathit{dependencies} = \{\mathit{neighbor} \mapsto 2, \mathit{myslave} \mapsto 2, \mathit{mymaster} \mapsto 0\}$$

To model a repair strategy in the traffic monitoring application, we introduce a simple type of repair actions:

$$\mathit{RepairActions} == \mathit{Dependency} \leftrightarrow (\mathit{Name} \times \mathit{Name})$$

Repair actions map dependencies to tuples of names. The first name in a tuple refers to the camera in the dependency, the second name indicates the new dependency in case the camera in the dependency fails.

A repair strategy model is defined as a set of repair actions:

$\mathit{RepairStrategy}$

$$\mathit{repairActions} : \mathit{RepairActions}$$

The repair strategies for the traffic case are defined as:

$\mathit{RepairStrategyOne}_{T2}$
$\mathit{RepairStrategy}$

$$\mathit{repairActions} = \{\mathit{neighbor} \mapsto (2, 3), \mathit{neighbormaster} \mapsto (3, 2)\}$$

The predicate states that if camera 2 fails the new neighbor of camera 1 will be camera 3, and if camera 3 fails, camera 2 will be the master its neighbor organization.

$\mathit{RepairStrategyTwo}_{T2}$
$\mathit{RepairStrategy}$

$$\mathit{repairActions} = \{\mathit{neighbor} \mapsto (1, 0), \mathit{neighbor} \mapsto (3, 0), \mathit{mymaster} \mapsto (3, 0)\}$$

$$
\begin{array}{|l}
\hline
\textit{RepairStrategyThree}_{T2} \\
\textit{RepairStrategy} \\
\hline
\textit{repairActions} = \{\textit{neighbor} \mapsto (2,1), \textit{neighbor} \mapsto (1,0), \\
\qquad \textit{myslave} \mapsto (2,0)\} \\
\hline
\end{array}
$$

The coordination model used for fault detection in the traffic monitoring application is defined as:

$$
\begin{array}{|l}
\hline
\textit{DependentNodes} \\
\textit{nodes} : \mathbb{P}\, \textit{Name} \\
\hline
\end{array}
$$

To define the coordination protocol used for fault detection, we introduce a simply type to represent time:

$\textit{Time} == \mathbb{N}$

The coordination protocol for fault detection is defined as:

$$
\begin{array}{|l}
\hline
\textit{PingEcho} \\
\\
\hline
\end{array}
$$

The above definitions enable us to define the coordination mechanism for fault detection:

$$
\begin{array}{|l}
\hline
\textit{PeerToPeer} \\
\textit{CoordinationMechanism}[\textit{PingEcho}, \textit{DependentNodes}, \textit{MessagePassing}] \\
\textit{pingTime} : \textit{Name} \leftrightarrow \textit{Time} \\
\textit{waitTime} : \textit{Time} \\
\hline
\mathrm{dom}\, \textit{pingTime} = \textit{model.nodes} \land \\
\forall\, n : \textit{model.nodes} \bullet \exists\, l : \textit{channel.links} \bullet \textit{first}(l) = n \\
\hline
\end{array}
$$

Ping time maps names to times. The domain of ping time are the nodes (cameras) in the dependency model. Ping time maintains the points in time when the last ping messages were sent to each of the cameras with a dependency. Wait time is a constant that indicates when an echo message should arrive after a ping messages has been sent. The last line of the predicate states that there are communication links available to each camera in the dependency model.

The concrete instance of the coordination mechanism for camera 1 at T2 is defined:

$$
\begin{array}{|l}
\hline
\textit{PeerToPeerOne}_{T2} \\
\textit{PeerToPeer} \\
\hline
\textit{model.nodes} = \{2, 3\} \\
\textit{channel.links} = \textit{traffic\_communication\_channel} \setminus \{1 \mapsto \textit{cam}_1\} \\
\textit{pingTime} = \{2 \mapsto 4430, 3 \mapsto 4440\} \\
\textit{waitTime} = 40 \\
\hline
\end{array}
$$

The predicate state that camera 1 has dependencies with camera 2 (its neighbor) and camera 3 (the master of its neighbor organization. The coordination mechanism has communication channels available to all the other cameras in the system. The last ping message was sent to camera 2 at time 4430 and to camera 3 at time 4440. Finally, the wait time for echo messages is 40 time units.

The instances of the coordination mechanisms for the other cameras at T2 are defined:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PeerToPeerTwo}_{T2}} \\
\hline
\mathit{PeerToPeer} \\
\hline
\mathit{model.nodes} = \{1, 3\} \\
\mathit{channel.links} = \mathit{traffic\_communication\_channel} \setminus \{2 \mapsto \mathit{cam}_2\} \\
\mathit{pingTime} = \{1 \mapsto 4432, 3 \mapsto 4434\} \\
\mathit{waitTime} = 40 \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PeerToPeerThree}_{T2}} \\
\hline
\mathit{PeerToPeer} \\
\hline
\mathit{model.nodes} = \{1, 2\} \\
\mathit{channel.links} = \mathit{traffic\_communication\_channel} \setminus \{3 \mapsto \mathit{cam}_3\} \\
\mathit{pingTime} = \{1 \mapsto 4436, 3 \mapsto 4440\} \\
\mathit{waitTime} = 40 \\
\hline
\end{array}$$

We can now define self-healing manager:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{SelfHealingManager}} \\
\hline
\mathit{Computation} \\
\mathit{coordinationMechanism} : \mathit{PeerToPeer} \\
\mathit{readDM} : \mathit{DependencyModel} \times \mathbb{P}\,\mathit{State} \rightarrow \mathbb{P}\,\mathit{State} \\
\mathit{writeDM} : \mathbb{P}\,\mathit{State} \times \mathit{DependencyModel} \rightarrow \mathit{DependencyModel} \\
\mathit{readRS} : \mathit{RepairStrategy} \times \mathbb{P}\,\mathit{State} \rightarrow \mathbb{P}\,\mathit{State} \\
\mathit{writeRS} : \mathbb{P}\,\mathit{State} \times \mathit{RepairStrategy} \rightarrow \mathit{RepairStrategy} \\
\mathit{sense} : \mathit{LocalTrafficMonitoringSystem} \times \mathbb{P}\,\mathit{State} \rightarrow \mathbb{P}\,\mathit{State} \\
\mathit{adapt} : \mathit{LocalTrafficMonitoringSystem} \times \mathbb{P}\,\mathit{State} \\
\qquad \rightarrow \mathit{LocalTrafficMonitoringSystem} \\
\mathit{send} : \mathbb{P}\,\mathit{State} \rightarrow \mathit{Message} \\
\mathit{receive} : \mathit{Message} \rightarrow \mathbb{P}\,\mathit{State} \\
\hline
\end{array}$$

A self-healing manager is a computation extended with a peer-to-peer coordination mechanism. Self-healing manager can act upon a dependency model and repair actions. It can sense a local traffic monitoring system and adapt it when a failure of a dependent camera is detected. Coordination with other self-healing managers is done using the exchange of messages.

The self-healing managers at T2 are defined:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{SelfHealingManagerOne}_{T2}} \\
\hline
\mathit{SelfHealingManager} \\
\mathit{PeerToPeerOne}_{T2} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{SelfHealingManagerTwo}_{T2}} \\
\hline
\mathit{SelfHealingManager} \\
\mathit{PeerToPeerTwo}_{T2} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{SelfHealingManagerThree}_{T2}} \\
\hline
\mathit{SelfHealingManager} \\
\mathit{PeerToPeerThree}_{T2} \\
\hline
\end{array}$$

A self-healing subsystem is defined as:

$$\begin{array}{l}
\textit{SelfHealingSubsystem} \\
\hline
\textit{dependencyModel} : \textit{DependencyModel} \\
\textit{repairStrategy} : \textit{RepairStrategy} \\
\textit{selfHealingManager} : \textit{SelfHealingManager} \\
\hline
\mathrm{dom}\ \textit{selfHealingManager.readDM} = \\
\quad \{(\textit{dependencyModel}, \textit{selfHealingManager.state})\} \land \\
\mathrm{dom}\ \textit{selfHealingManager.writeDM} = \\
\quad \{(\textit{selfHealingManager.state}, \textit{dependencyModel})\} \land \\
\mathrm{dom}\ \textit{selfHealingManager.readRS} = \\
\quad \{(\textit{repairStrategy}, \textit{selfHealingManager.state})\} \land \\
\mathrm{dom}\ \textit{selfHealingManager.writeRS} = \\
\quad \{(\textit{selfHealingManager.state}, \textit{repairStrategy})\} \land \\
\mathrm{dom}\ \textit{selfHealingManager.send} = \{\textit{selfHealingManager.state}\} \land \\
\forall\, \textit{dependency} : \textit{dependencyModel.dependencies} \bullet \exists\, l : \\
\quad \textit{selfHealingManager.coordinationMechanism.channel.links}; \\
\qquad d : \textit{Dependency};\ n : \textit{Name} \bullet \textit{dependency} = (d, n) \land \textit{first}(l) = n \land \\
\forall\, \textit{repairAction} : \textit{repairStrategy.repairActions} \bullet \exists\, ol, nl : \\
\quad \textit{selfHealingManager.coordinationMechanism.channel.links}; \\
\qquad d : \textit{Dependency};\ on, nn : \textit{Name} \bullet \\
\qquad\quad \textit{repairAction} = (d, (on, nn)) \land \textit{first}(ol) = on \land \textit{first}(nl) = nn
\end{array}$$

The predicate states that a self-healing manager can only act upon the local dependency model and repair strategy. Messages for coordination are produced based on the current local state of the computation. Furthermore, the predicate states that there is a communication link with every camera with a dependency and with every camera in any of the repair actions.

The concrete self-healing subsystem for camera 1 at T2 is defined:

$$\begin{array}{l}
\textit{SelfHealingSubsystemOne}_{T2} \\
\hline
\textit{SelfHealingSubsystem} \\
\textit{DependencyModelOne}_{T2} \\
\textit{RepairStrategyOne}_{T2} \\
\textit{SelfHealingManagerOne}_{T2}
\end{array}$$

The self-healing sybsystems for the other cameras at T2 are defined:

$$\begin{array}{l}
\textit{SelfHealingSubsystemTwo}_{T2} \\
\hline
\textit{SelfHealingSubsystem} \\
\textit{DependencyModelTwo}_{T2} \\
\textit{RepairStrategyTwo}_{T2} \\
\textit{SelfHealingManagerTwo}_{T2}
\end{array}$$

$$\begin{array}{l}
\textit{SelfHealingSubsystemThree}_{T2} \\
\hline
\textit{SelfHealingSubsystem} \\
\textit{DependencyModelThree}_{T2} \\
\textit{RepairStrategyThree}_{T2} \\
\textit{SelfHealingManagerThree}_{T2}
\end{array}$$

Finally, we model a timeout of a ping message. First, we introduce a simple clock:

$$\begin{array}{|l}\hline \textit{Clock} \\ \textit{time} : \textit{Time} \\ \hline\end{array}$$

The clock at T2 is defined:

$$\begin{array}{|l}\hline \textit{Clock}_{T3} \\ \textit{Clock} \\ \hline \textit{time} = 4444 \\ \hline\end{array}$$

Time passes by as follows:

$$\begin{array}{|l}\hline \textit{Tick} \\ \Delta \textit{Clock} \\ \hline \textit{time}' = \textit{time} + 1 \\ \hline\end{array}$$

A timeout is defined as:

$$\begin{array}{|l}\hline \textit{Timeout} \\ \Xi \textit{SelfHealingManager} \\ \textit{Tick} \\ n! : \textit{Name} \\ \hline \exists\, n! : \textit{Name};\ t : \textit{Time} \bullet \\ \qquad (n!, t) \in \textit{coordinationMechanism.pingTime} \wedge \\ \qquad t + \textit{coordinationMechanism.waitTime} > \textit{time}' \\ \hline\end{array}$$

The schema tells us that a timeout of a self-healing manager does not change its state. A timeout happens when the clock makes a tick. The predicate states that a timeout for a particular camera is reached when the time after the tick exceeds the last ping time for that camera plus the wait time.

The timeout for self-healing manager 1 after the crash of camera 2 is defined as:

$$\begin{array}{|l}\hline \textit{Timeout}_1 \\ \textit{Timeout} \\ \Xi \textit{SelfHealingManagerOne}_{T2} \\ \textit{Tick} \\ n! : \textit{Name} \\ \hline \textit{time} = 4470 \\ n! = 2 \\ \hline\end{array}$$

The timeout happens when the clock makes a tick at time “4470” (recall that the ping message to camera 2 was sent at time “4430” and the waiting time is 40 time units). The timeout applies for camera 2.

D.4 Traffic Jam Monitoring System

To define a traffic jam monitoring system, we first define a local camera system:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{LocalCameraSystem}} \\
\hline
\mathit{localTrafficMonitoringSystem} : \mathit{LocalTrafficMonitoringSystem} \\
\mathit{selfHealingSubsystem} : \mathit{SelfHealingSubsystem} \\
\mathit{myName} : \mathit{Name} \\
\hline
\mathrm{dom}\ \mathit{selfHealingSubsystem}.\mathit{selfHealingManager}.\mathit{sense} = \\
\qquad \{(\mathit{localTrafficMonitoringSystem}, \mathit{selfHealingSubsystem}.\mathit{selfHealingManager}.\mathit{state})\} \land \\
\mathrm{dom}\ \mathit{selfHealingSubsystem}.\mathit{selfHealingManager}.\mathit{adapt} = \\
\qquad \{(\mathit{localTrafficMonitoringSystem}, \mathit{selfHealingSubsystem}.\mathit{selfHealingManager}.\mathit{state})\} \\
\hline
\end{array}$$

A local camera system consists of a local traffic monitoring system that deals with traffic jam monitoring, and a self-healing subsystem that deals with failure management. A local camera system has a unique name that is used for communication.

The concrete local camera systems at T2 are defined:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{LocalCameraSystemOne}_{T2}} \\
\hline
\mathit{LocalCameraSystem} \\
\mathit{LocalTrafficMonitoringSystemOne}_{T2} \\
\mathit{SelfHealingSubsystemOne}_{T2} \\
\hline
\mathit{myName} = 1 \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{LocalCameraSystemTwo}_{T2}} \\
\hline
\mathit{LocalCameraSystem} \\
\mathit{LocalTrafficMonitoringSystemTwo}_{T2} \\
\mathit{SelfHealingSubsystemTwo}_{T2} \\
\hline
\mathit{myName} = 2 \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{LocalCameraSystemThree}_{T2}} \\
\hline
\mathit{LocalCameraSystem} \\
\mathit{LocalTrafficMonitoringSystemThree}_{T2} \\
\mathit{SelfHealingSubsystemThree}_{T2} \\
\hline
\mathit{myName} = 3 \\
\hline
\end{array}$$

A situated local camera system is defined as:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{SituatedLocalCameraSystem}} \\
\hline
\mathit{TrafficEnvironment} \\
\mathit{LocalCameraSystem} \\
\mathit{context} : \mathit{Context} \\
\hline
\mathit{context} \subseteq \mathit{attributes} \land \\
\mathrm{dom}(\mathit{localTrafficMonitoringSystem}.\mathit{computation}.\mathit{perceive}) = \\
\qquad \{\mathit{attrs} : \mathit{Context} \mid \mathit{attrs} \subseteq \mathit{context} \bullet \\
\qquad (\mathit{localTrafficMonitoringSystem}.\mathit{computation}.\mathit{state}, \mathit{attrs})\} \land \\
\mathit{dom}\ (\mathit{localTrafficMonitoringSystem}.\mathit{computation}.\mathit{effect}) = \\
\qquad \{\mathit{attrs} : \mathit{Context} \mid \mathit{attrs} \subseteq \mathit{context} \bullet \\
\qquad (\mathit{localTrafficMonitoringSystem}.\mathit{computation}.\mathit{state}, \mathit{attrs})\} \\
\hline
\end{array}$$

A situated local camera system is a local camera system that is situated in a traffic environment. A situated local camera system's access to the environment is restricted to the context in which the camera is situated.

The concrete situated local camera 1 in the example is defined at T2 as:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{SituatedLocalCameraSystemOne}_{T2}} \\
\mathit{TrafficEnvironment}_{T2} \\
\mathit{LocalCameraSystemOne}_{T2} \\
\mathit{context} : \mathit{Context} \\
\hline
\mathit{context} = \{\mathit{camera}_2\ , \mathit{camera}_3\ , \mathit{freeflow\_zone}_1\} \\
\hline
\end{array}
$$

The context of camera 1 consists of the two other cameras in the system and the traffic in its viewing range.

The other concrete situated local cameras are defined:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{SituatedLocalCameraSystemTwo}_{T2}} \\
\mathit{TrafficEnvironment}_{T2} \\
\mathit{LocalCameraSystemTwo}_{T2} \\
\mathit{context} : \mathit{Context} \\
\hline
\mathit{context} = \{\mathit{camera}_1\ , \mathit{camera}_3\ , \mathit{congested\_zone}_2\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{SituatedLocalCameraSystemThree}_{T2}} \\
\mathit{TrafficEnvironment}_{T2} \\
\mathit{LocalCameraSystemThree}_{T2} \\
\mathit{context} : \mathit{Context} \\
\hline
\mathit{context} = \{\mathit{camera}_1\ , \mathit{camera}_2\ , \mathit{congested\_zone}_3\} \\
\hline
\end{array}
$$

We can now define a traffic jam monitoring system:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{TrafficJamMonitoringSystem}} \\
\mathit{localCamaraSystems} : \mathbb{P}\ \mathit{SituatedLocalCameraSystem} \\
\hline
\forall\ \mathit{lcs} : \mathit{localCamaraSystems};\ \mathit{msgs} : \mathbb{P}\ \mathit{Message};\ \mathit{addressees} : \mathbb{P}\ \mathit{Name} \bullet \\
\qquad \mathit{msgs} = \mathit{ran}\ (\mathit{lcs}.\mathit{localTrafficMonitoringSystem}.\mathit{computation}.\mathit{send}) \land \\
\qquad \mathit{addressees} = \{n : \mathit{Name};\ \mathit{msg} : \mathit{msgs} \mid n = \mathit{msg}.\mathit{from} \bullet n\} \land \\
\qquad \mathit{addressees} = \mathit{dom}\ (\mathit{lcs}.\mathit{localTrafficMonitoringSystem}.\mathit{computation}. \\
\qquad\qquad \mathit{trafficCoordinationMechanism}.\mathit{channel}.\mathit{links}) \land \\
\forall\ \mathit{lcs} : \mathit{localCamaraSystems};\ d : \mathit{Dependency};\ n : \mathit{Name} \bullet \\
\qquad (d, n) \in \mathit{lcs}.\mathit{selfHealingSubsystem}.\mathit{dependencyModel}.\mathit{dependencies} \land \\
\qquad n \neq \mathit{lcs}.\mathit{myName} \land \\
\forall\ \mathit{lcs} : \mathit{localCamaraSystems};\ \mathit{shmsgs} : \mathbb{P}\ \mathit{Message};\ \mathit{shaddressees} : \mathbb{P}\ \mathit{Name} \bullet \\
\qquad \mathit{shmsgs} = \mathit{ran}\ (\mathit{lcs}.\mathit{selfHealingSubsystem}.\mathit{selfHealingManager}.\mathit{send}) \land \\
\qquad \mathit{shaddressees} = \{n : \mathit{Name};\ \mathit{msg} : \mathit{msgs} \mid n = \mathit{msg}.\mathit{from} \bullet n\} \land \\
\qquad \mathit{shaddressees} = \mathit{dom}\ (\mathit{lcs}.\mathit{selfHealingSubsystem}.\mathit{selfHealingManager}. \\
\qquad\qquad \mathit{coordinationMechanism}.\mathit{channel}.\mathit{links}) \\
\hline
\end{array}
$$

A traffic monitoring system consists of a set of situated local camera systems. The first part of the predicate defines the scope of communication of the base-level subsystems. The second part defines the dependencies in the system and states that a local camera system cannot depend on itself. The third part of the predicate defines the scope of communication of the self-healing subsystems.

At T2 the state of the traffic jam monitoring system is:

$$\begin{array}{|l}
\hline
\textit{TrafficJamMonitoringSystem}_{T2} \\
\hline
\textit{TrafficJamMonitoringSystem} \\
\textit{SituatedLocalCameraSystemOne}_{T2} \\
\textit{SituatedLocalCameraSystemTwo}_{T2} \\
\textit{SituatedLocalCameraSystemThree}_{T2} \\
\hline
\end{array}$$

Note that we have not provided the specification of the situated local camera system of cameras 2 and 3 in this document. For the omitted part of the specification, we refer the interested reader to [Weyns et al. 2010b].

At T3 when camera 2 fails, the state of the traffic camera system is changed as follows:

$$\begin{array}{|l}
\hline
\textit{TrafficJamMonitoringSystem}_{T3} \\
\hline
\Delta\,\textit{TrafficJamMonitoringSystem}_{T2} \\
\textit{lcs}2? : \textit{SituatedLocalCameraSystem} \\
\hline
\textit{lcs}2? \in \textit{localCamaraSystems} \wedge \\
\textit{lcs}2?.\textit{myName} = 2 \wedge \\
\textit{localCamaraSystems}' = \textit{localCamaraSystems} \setminus \{\textit{lcs}2?\} \\
\hline
\end{array}$$

To conclude, we formalize how camera 1 recovers from the failure of camera 2 that happens after the time out of the ping message. First we define two helper functions to update the different parts of the camera system:

$$\begin{array}{|l}
\textit{adaptLocalTrafficMonitoringSystem} : \textit{SituatedLocalCameraSystem} \times \textit{Attribute} \times \\
\quad \textit{EnvironmentRepresentation} \times \textit{Name} \rightarrow \textit{LocalTrafficMonitoringSystem} \\
\hline
\forall\, \textit{slcs} : \textit{SituatedLocalCameraSystem};\ \textit{ultms} : \textit{LocalTrafficMonitoringSystem}; \\
\quad \textit{camera} : \textit{Attribute};\ \textit{cam} : \textit{EnvironmentRepresentation};\ n : \textit{Name} \bullet \\
\textit{ultms}.\textit{trafficModel}.\textit{representations} = \\
\quad \textit{slcs}.\textit{localTrafficMonitoringSystem}.\textit{trafficModel}.\textit{representations} \setminus \{\textit{cam}\} \wedge \\
\textit{ultms}.\textit{trafficModel}.\textit{mapping} = \\
\quad \textit{slcs}.\textit{localTrafficMonitoringSystem}.\textit{trafficModel}.\textit{mapping} \setminus \{\{\textit{camera}\} \mapsto \textit{cam}\} \wedge \\
\textit{ultms}.\textit{computation}.\textit{trafficCoordinationMechanism}.\textit{orgProtocol}.\textit{role} = \\
\quad \textit{slcs}.\textit{localTrafficMonitoringSystem}.\textit{computation}. \\
\qquad \textit{trafficCoordinationMechanism}.\textit{orgProtocol}.\textit{role} \wedge \\
\textit{ultms}.\textit{computation}.\textit{trafficCoordinationMechanism}.\textit{orgModel}.\textit{partners} = \\
\quad \textit{slcs}.\textit{localTrafficMonitoringSystem}.\textit{computation}. \\
\qquad \textit{trafficCoordinationMechanism}.\textit{orgModel}.\textit{partners} \setminus \{n\} \wedge \\
\textit{ultms}.\textit{computation}.\textit{trafficCoordinationMechanism}.\textit{orgModel}.\textit{neighborOrganizations} = \\
\quad \textit{slcs}.\textit{localTrafficMonitoringSystem}.\textit{computation}. \\
\qquad \textit{trafficCoordinationMechanism}.\textit{orgModel}.\textit{neighborOrganizations} \wedge \\
\textit{ultms}.\textit{computation}.\textit{trafficCoordinationMechanism}.\textit{channel}.\textit{links} = \\
\quad \textit{slcs}.\textit{localTrafficMonitoringSystem}.\textit{computation}. \\
\qquad \textit{trafficCoordinationMechanism}.\textit{channel}.\textit{links} \setminus \{n \mapsto \textit{cam}\} \wedge \\
\textit{adaptLocalTrafficMonitoringSystem}(\textit{slcs}, \textit{camera}, \textit{cam}, n) = \textit{ultms}
\end{array}$$

The first helper function takes a situated local camera system and the data of a camera that fails and returns the adapted local traffic monitoring system of the camera system. The function is applicable for situations in which a neighboring camera fails that plays the role of slave. The adaptation includes:

—The representation of the camera is removed from the set of representations;

—The mapping of the representation to the real camera is removed;

—The role of the traffic monitoring system is not changed;

—The failing camera is removed from the list of partners;

—The neighbor organizations are not changed (the failing camera is a slave of a neighbor organization);

—The communication link to the failing camera is removed.

$$
\begin{array}{l}
\mathit{updateSelfHealingSubsystem} : \mathit{SituatedLocalCameraSystem} \times \mathit{Attribute} \times \\
\quad \mathit{EnvironmentRepresentation} \times \mathit{Name} \rightarrow \mathit{SelfHealingSubsystem} \\
\hline
\forall\, \mathit{slcs} : \mathit{SituatedLocalCameraSystem};\ \mathit{ushs} : \mathit{SelfHealingSubsystem}; \\
\quad \mathit{camera} : \mathit{Attribute};\ \mathit{cam} : \mathit{EnvironmentRepresentation};\ n : \mathit{Name} \bullet \\
\exists\, \mathit{newneighbor} : \mathit{Name} \bullet \mathit{slcs}.\mathit{selfHealingSubsystem}.\mathit{repairStrategy}. \\
\quad \mathit{repairActions} \rhd \{(n, \mathit{newneighbor})\} = \{\mathit{neighbor} \mapsto (n, \mathit{newneighbor})\} \wedge \\
\mathit{ushs}.\mathit{dependencyModel}.\mathit{dependencies} = \\
\quad \mathit{slcs}.\mathit{selfHealingSubsystem}.\mathit{dependencyModel}.\mathit{dependencies} \\
\quad\quad \oplus \{\mathit{neighbor} \mapsto \mathit{newneighbor}\} \wedge \\
\mathit{ushs}.\mathit{repairStrategy}.\mathit{repairActions} = \\
\quad \mathit{slcs}.\mathit{selfHealingSubsystem}.\mathit{repairStrategy}.\mathit{repairActions} \setminus \\
\quad\quad \{\mathit{neighbor} \mapsto (n, \mathit{newneighbor})\} \wedge \\
\mathit{ushs}.\mathit{selfHealingManager}.\mathit{state} = \\
\quad \mathit{slcs}.\mathit{selfHealingSubsystem}.\mathit{selfHealingManager}.\mathit{state} \wedge \\
\mathit{ushs}.\mathit{selfHealingManager}.\mathit{coordinationMechanism}.\mathit{protocol} = \\
\quad \mathit{slcs}.\mathit{selfHealingSubsystem}.\mathit{selfHealingManager}.\mathit{coordinationMechanism}. \\
\quad\quad \mathit{protocol} \wedge \\
\mathit{ushs}.\mathit{selfHealingManager}.\mathit{coordinationMechanism}.\mathit{model}.\mathit{nodes} = \\
\quad \mathit{slcs}.\mathit{selfHealingSubsystem}.\mathit{selfHealingManager}.\mathit{coordinationMechanism}. \\
\quad\quad \mathit{model}.\mathit{nodes} \setminus \{n\} \wedge \\
\mathit{ushs}.\mathit{selfHealingManager}.\mathit{coordinationMechanism}.\mathit{channel}.\mathit{links} = \\
\quad \mathit{slcs}.\mathit{selfHealingSubsystem}.\mathit{selfHealingManager}.\mathit{coordinationMechanism}. \\
\quad\quad \mathit{channel}.\mathit{links} \setminus \{n \mapsto \mathit{cam}\} \wedge \\
\exists\, \mathit{pt} : \mathit{Time} \bullet \{n\} \lhd \mathit{slcs}.\mathit{selfHealingSubsystem}.\mathit{selfHealingManager}. \\
\quad \mathit{coordinationMechanism}.\mathit{pingTime} = \{(n \mapsto \mathit{pt})\} \wedge \\
\mathit{ushs}.\mathit{selfHealingManager}.\mathit{coordinationMechanism}.\mathit{pingTime} = \\
\quad \mathit{slcs}.\mathit{selfHealingSubsystem}.\mathit{selfHealingManager}.\mathit{coordinationMechanism}. \\
\quad\quad \mathit{pingTime} \setminus \{n \mapsto \mathit{pt}\} \wedge \\
\mathit{ushs}.\mathit{selfHealingManager}.\mathit{coordinationMechanism}.\mathit{waitTime} = \\
\quad \mathit{slcs}.\mathit{selfHealingSubsystem}.\mathit{selfHealingManager}.\mathit{coordinationMechanism}. \\
\quad\quad \mathit{waitTime} \wedge \\
\mathit{updateSelfHealingSubsystem}(\mathit{slcs}, \mathit{camera}, \mathit{cam}, n) = \mathit{ushs}
\end{array}
$$

The second helper function updates the self-healing system after a camera fails. This function is applicable for the same type of situations as the first helper function. The update includes:

—The dependencies are updated with the new neighbor;

—The repair actions related to the crashed camera are removed.

—The computation state of the self-healing manager is not changed;

—The coordination protocol is not changed;

—The node of the failing camera is removed from the coordination model;

—The communication link to the failing camera is removed;

—The ping time to the failing camera is removed;

—The wait time for ping messages is not changed.

Finally, the recovery is defined as:

$$
\begin{array}{|l}
\hline
\textit{CameraOneRecoversFromFailureCameraTwo} \\
\Delta\, \textit{TrafficJamMonitoringSystem}_{T3} \\
\textit{TrafficEnvironment}_{T3} \\
\textit{Timeout}_1 \\
lcs1?, lcs1! : \textit{SituatedLocalCameraSystem} \\
camera : \textit{Attribute} \\
cam : \textit{EnvironmentRepresentation} \\
n : \textit{Name} \\
\hline
\{camera\} = \textit{first}(c?) \wedge \\
\textit{traffic\_attribute\_representation\_mapping} = \\
\quad \textit{traffic\_attribute\_representation\_mapping} \setminus \{\{camera\} \mapsto cam\} \wedge \\
\textit{traffic\_communication\_channel} = \\
\quad \textit{traffic\_communication\_channel} \setminus \{n \mapsto cam\} \wedge \\
lcs1? \in \textit{localCamaraSystems} \wedge \\
lcs1?.myName = 1 \wedge \\
lcs1!.myName = lcs1?.myName \wedge \\
lcs1!.context = lcs1?.context \setminus \{camera\} \wedge \\
lcs1!.\textit{selfHealingSubsystem} = \\
\quad \textit{updateSelfHealingSubsystem}(lcs1?, camera, cam, n) \wedge \\
lcs1!.\textit{localTrafficMonitoringSystem} = \\
\quad \textit{adaptLocalTrafficMonitoringSystem}(lcs1?, camera, cam, n) \wedge \\
\textit{localCamaraSystems}' = \textit{localCamaraSystems} \setminus \{lcs1?\} \cup \{lcs1!\} \\
\hline
\end{array}
$$

The specification *declaratively* specifies what state of the local camera system is adapted after the failure of the camera. The first part of the predicate assigns the attribute, representation, and name of the failing camera to the variables camera, cam, and n, using the camera failure event. Next, the attribute representation mappings and communication channels are updated; the recovering local camera system is selected (with myName = 1) and the failing camera is removed from its context. Then adaptation is specified, consisting of two parts: an update of the state of the self-healing subsystem and the actual adaptation of the local traffic monitoring system (using the helper functions defined above). From an *operational* point of view, the self-healing manager will update its state and apply the adaptation of the local traffic monitoring system using various read and write operations. An analogous specification can be defined for the recovery of camera 3 in the scenario.

E. IBM AUTONOMIC MANAGER FRAMEWORK

In this section, we study the model for IBM's autonomic manager framework [IBM 2006], which advocates a hierarchical composition of autonomic managers. This case illustrates modeling with the primitives from FORMS's Reflection and MAPE-K perspectives. The basic building block is the autonomic manager that implements a control loop (MAPE-K). The autonomic control loop consists of four basic activities: monitor, analyze, plan, and execute. The activities share knowledge that typically includes a model of the managed element(s) and a description of goals [Huebscher and McCann. 2008]. An autonomic manager provides sensors and effectors for other autonomic managers to use. As further detailed below, this enables hierarchical composition of autonomic managers.

Figure 10 describes the autonomic manager using the basic FORMS's primitives. The *autonomic manager* corresponds to a *self-adaptive unit* from FORMS. An *autonomic manager* com-

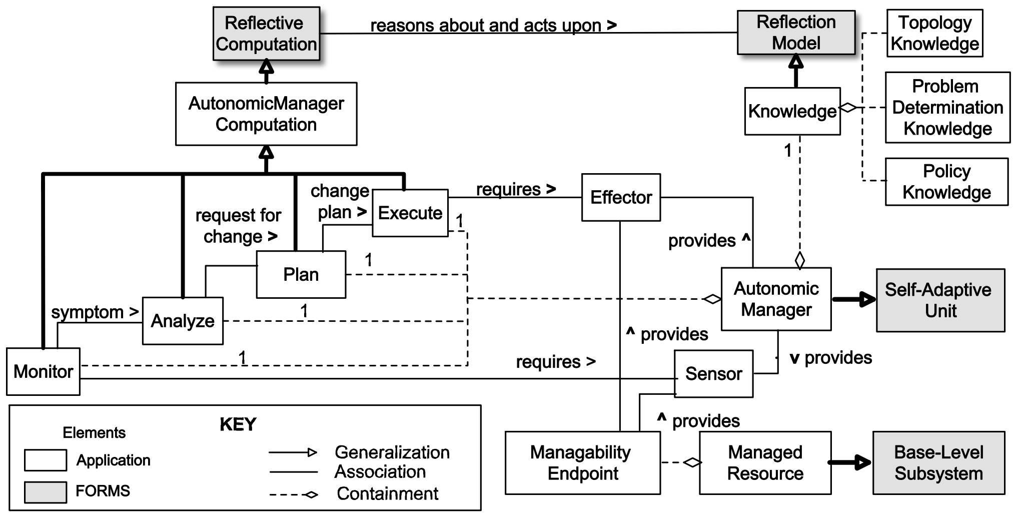


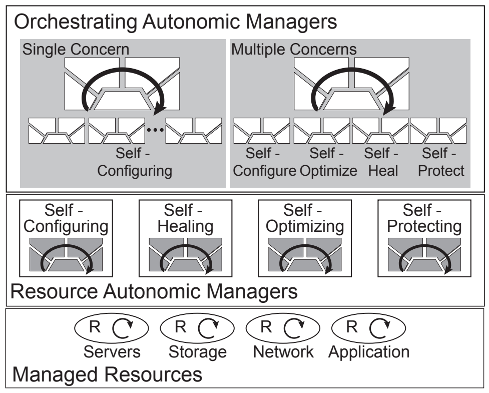
Fig. 10. MAPE-K's computations and knowledge in relation to FORMS primitives. White boxes represent IBM's autonomic manager constructs, gray boxes represent FORMS constructs.

prises four types of *autonomic manager computations*, which instantiate four concrete types of *reflective computation* from FORMS: *monitor*, *analyze*, *plan*, and *execute*. *Autonomic manager* components can reason about and act upon the shared *knowledge*, which instantiates *reflection model* from FORMS. *Monitor* requires a *sensor* to monitor the managed element, which can be either a *managed resource* (corresponding to FORMS's *base-level subsystem*) or an *autonomic manager* (corresponding to FORMS's *self-adaptive unit*). In the former case, the *sensor* is provided by the *manageability endpoint* of the *managed resource*. *Execute* requires an *effector* to adapt the managed element according to the plans constructed by the plan component.

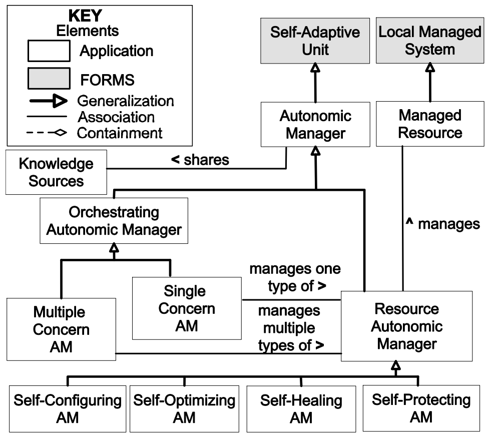
While there is a shared understanding on the different types of computations in a MAPE-K autonomic manager, the role of *knowledge* is less clear. According to [Kephart and Chess 2003], *knowledge* refers to the data collected from *managed resources*, models for analysis such as queueing network models, policy information, and action plans. [Miller 2005] groups the different forms of knowledge in three distinct types: *topology knowledge*, *policy knowledge*, and *problem determination knowledge*.

We now briefly explain how hierarchies of autonomic systems are constructed using *autonomic managers* and modeled in FORMS. Figure 11(a) shows some different types of autonomic managers [Kephart and Chess 2003], arranged in a hierarchy.

Figure 11(b) depicts how FORMS primitives are used to model the hierarchy in Figure 11(a). A *resource autonomic manager* manages a *managed resource*. Four concrete types are distinguished: managers for *self-configuring*, *self-healing*, *self-optimizing*, and *self-protecting*. *Orchestrating autonomic managers* on the other hand manage groups of *resource autonomic managers*. In particular, a *single concern orchestrating autonomic manager* manages a group of *resource autonomic managers* of the same type, while a *multiple concern orchestrating autonomic manager* manages a group of *resource autonomic managers* of different type. *Orchestrating autonomic managers* themselves can be managed by higher-level *autonomic managers*, just like a *self-adaptive unit* in FORMS that can be reflected upon by another *self-adaptive unit* from the level above. A hierarchy of *autonomic managers* thus corresponds to the reflective levels in FORMS. In a hierarchy of *autonomic managers*, data can be obtained and shared via *knowledge sources*. According to [Miller 2005], a *knowledge source* is an implementation of



(a) Hierarchy of Autonomic Managers



(b) Hierarchy using FORMS primitives

Fig. 11. Autonomic manager hierarchies in FORMS. White boxes represent autonomic manager constructs, gray boxes represent FORMS constructs.

a registry, dictionary, database or other repository that provides access to knowledge that need to be shared among autonomic managers. Appendix C.3 provides a formal specification of the elements shown in Figure 11(b) and uses them to describe a simple example of a self-adaptive autonomic system.

F. SENSOR NETWORK SYSTEM

The final self-adaptive software system that we study in light of FORMS is MIDAS [Malek et al. 2007], which is an application family developed in collaboration between one of the authors and Bosch engineers. MIDAS is a security monitoring distributed application composed of a large number of wirelessly connected sensors, gateways, hubs, and PDAs. The sensors are used to monitor the environment around them, and communicate their status to one another and to the gateways. The gateway nodes are responsible for managing and coordinating the sensors. Furthermore, the gateways translate, aggregate, and fuse the data received from the sensors, and propagate the appropriate data (e.g., events) to the hubs. Hubs, in turn, are used to evaluate and visualize the sensor data for human users, as well as to provide an interface through which a user can send control commands to various sensors and gateways in the system. Hubs may also be configured to propagate the appropriate sensor data to PDAs, which are used by the mobile users of the systems.

MIDAS has several QoS requirements that need to be satisfied in tandem, in particular response time and energy consumption. The engineers found the deployment of software components to hardware devices (i.e., deployment architecture) to have a significant impact on these two QoS requirements. For instance, the availability of local communication on a sensor platform reduces the response time, but would potentially increase the rate at which the sensor's battery power is drained. As a result, a self-adaptation framework was developed that in response to changes in system properties (e.g., changes in remaining battery, fluctuations in network bandwidth) looks for the near-optimal deployment architecture at runtime and improves the system's QoS through redeployment of its software components.

Figure 12 shows the framework's distributed instantiation. Each host runs an instance of the framework that consists of the following components: *Deployment Analyzer*— maintains a representation of the system's deployment architecture, such as the hardware hosts, software components, component dependencies, various system parameters of interest (e.g., network

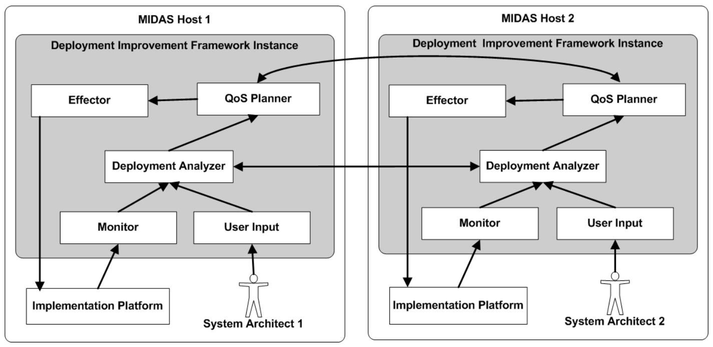


Fig. 12. Deployment self-optimization approach in MIDAS case study.

bandwidth, frequency of interactions), and uses this model to assess the system's current QoS attributes. *QoS Planner*—searches for a deployment architecture that improves the QoS attributes and performs a trade-off analysis to ensure that the cost of redeployment (e.g., resource overhead and temporary unavailability of components) does not exceed the benefits (e.g., improvements in QoS) of it. *Monitor*—collects data on changes in system parameters and identifies patterns of change to initiate adaptation. *Effector*—adapts the system through migration of its component. *User Input*—used by the system's user to input their QoS preferences in the form of a utility function, as well as the information about system parameters that may not be easily monitored (e.g., security of a network link). The framework described above has been realized using an integration of several tools; the interested reader may find more details at [Malek et al. 2007].

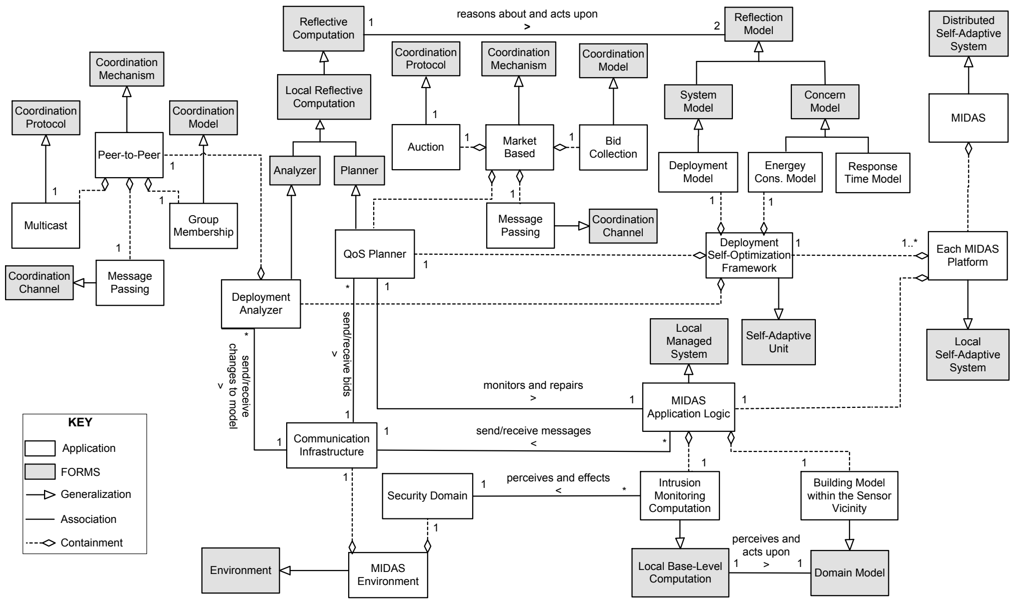


Fig. 13. Precise specification of MIDAS through FORMS constructs. White boxes represent MIDAS constructs, gray boxes represent FORMS constructs.

Figure 13 shows the specification of MIDAS using FORMS. Similar to the previous two case studies we are able to precisely define the elements of MIDAS by extending the FORMS constructs. Unlike the traffic monitoring case study that has a single concern, in MIDAS we have two QoS objectives (concerns), *energy consumption* and *response time*, which extend FORMS's *concern model*. MIDAS's *deployment model*, which represents the allocation of software components to hardware platforms, extends FORMS's *system model* . Deployment model is used by MIDAS's reflective computations for making adaptation decisions.

Another important difference in this case study is the existence of two different types of reflective computations, *QoS planner* and *deployment analyzer*, which are accompanied by two types of coordination mechanism, *market-based* and *peer-to-peer*. As shown in Figure 12, *QoS planner* runs on every MIDAS platform and coordinates with other *QoS planners* via an *auction protocol*, which as specified in Figure 13, corresponds to FORMS's *coordination protocol*. In this protocol, each planner periodically initiates an auction for one of its locally deployed components, which allows other planner components to participate by placing bids. Each bid contains a *utility* value that corresponds to the improvements in system's overall QoS as a result of redeploying the auctioned component to the bidding device. *Bid collection* represents the *coordination model* and consists of tuples of form (bidder id, utility value) maintained by each *QoS planner*. The format of tuples maintained may change depending on the type of auction (e.g., Vickrey, Dutch, English).

Deployment analyzer is another type of reflective computation that just like *QoS planner* exists on every MIDAS platform. In order to quantitatively estimate the current QoS obtained for the system's deployment, each *deployment analyzer* needs to first augment its local model with the information available from the neighboring *deployment analyzers*. For this purpose, each *Deployment analyzer* uses a *peer-to-peer* coordination mechanism in which each peer *multi-casts* (FORMS's *coordination protocol*) changes in its local model to the peers within its group membership (FORMS's *coordination model*).

As depicted in Figure 13 we were able to use FORMS to precisely specify other architectural facets of MIDAS (e.g., separation of reflective subsystem from managed subsystem) that for brevity are not discussed further here.