FORMS: a FOrmal Reference Model for Self-adaptation

Danny Weyns
Depart. of Computer Science
Katholieke Universiteit
Leuven, Belgium
danny.weyns@cs.kuleuven.be

Sam Malek
Depart. of Computer Science
George Mason University,
Washington, USA
smalek@gmu.edu

Jesper Andersson
Depart. of Computer Science
Linnaeus University,
Växjö, Sweden
jesper.andersson@lnu.se

ABSTRACT

Self-adaptive software systems are an emerging class of systems that adjust their behavior at runtime to achieve certain functional or quality of service objectives. The construction of such systems has shown to be significantly more challenging than traditional systems, partly because researchers and practitioners have been struggling with the lack of a precise method of describing, comparing, and evaluating alternative architectural choices. In this paper, we introduce a reference model, entitled FOrmal Reference Model for Self-adaptation (FORMS), which intends to alleviate this pressing issue. FORMS consists of a small number of formally specified modeling primitives that correspond to the key variation points within self-adaptive software systems, and a set of relationships that guide their composition. We present our experiences with applying FORMS to several existing systems, which not only demonstrates its ability to precisely illuminate their underlying characteristics, but also its potential as a method of rigorously specifying architectural patterns for this domain.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design, theory

1. INTRODUCTION

Contemporary software systems have become increasingly more complex, and continue to pervade every facet of our day-to-day activities with no signs of abating. At the same time, the cost and effort associated with the unwieldy methods of maintaining and updating software after it is deployed have become unacceptable. This has called for the development of self-adaptive software systems. Self-adaptive software systems are capable of autonomously changing their behavior to achieve certain functional or quality of service (QoS) objectives [4, 12, 13].

The development of self-adaptive software systems has shown to be significantly more challenging than traditional systems [4], and over the past decade numerous solutions for alleviating the situation have been developed [12, 10, 16]. In particular, researchers and practitioner have proposed several frameworks for constructing such systems. Some have aimed to serve as a conceptual guideline, such as IBM's MAPE-K [12] that describes the different stages of self-adaptation, and Mary Shaw's work [18] that recognizes the feedback-control loop as an essential component of any self-adaptive system. Others have adopted an implementation perspective, such as Rainbow [10] and Archstudio [16], both of which advocate a software architecture-based approach for assessing the adaptation decisions and making the changes.

All of these frameworks are intended to serve merely as guidelines, and provide significant leeway in how the engineer designs and constructs 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 expressive and formal enough to clearly distinguish between the alternative designs, nor is that their intended use.

The hallmark of any established engineering field is the ability to precisely express the design choices and evaluate the alternatives. At the same time, there is a lack of rigorous understanding of the primitive constructs and principles that shape a self-adaptive software system [2], which we believe is hindering further progress in this area. In this paper, we present FORMS, short for FOrmal Reference Model for Self-adaptation, which enables the engineers to effectively describe, study, and evaluate alternative design choices for self-adaptive software systems. FORMS builds on the existing frameworks and established principles, such as MAPE-K [12], computational reflection [14], and architecture-based adaptation [13, 16]. The reference model 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 compare alternative solutions.

Since there are many types of self-adaptive software, developing a one-size-fits-all reference model that can represent every aspect of such systems is not practical. Thus, our objective in the development of FORMS has been to make it extensible, without compromising precision and expressiveness. Through an extensive study of the literature, we first developed the minimum set of primitives necessary for formally specifying a self-adaptive system, albeit at a high-level of abstraction. Applying the high-level model to several existing systems resulted in further refinement and specialization of the primitives. This exercise helped us to demonstrate the reference model's ability to illuminate the key, in some cases hidden, characteristics of these systems. The primitives refined in this manner

ICAC'10, June 7–11, 2010, Washington, DC, USA.

enable the engineers to derive and document a catalog of architectural patterns (i.e., known solutions) for different domains.

The remainder of this paper is organized as follows. Section 2 defines the high-level reference model using Z notation, and illustrate the concepts via an intuitive example. Section 3 presents our experiences with using the high-level reference model in two case studies. Our findings from these studies are then used to refine the model by extending it with further details, which we describe in Section 4. Finally, the paper concludes with an overview of related work, applications and contributions of FORMS, and future avenues of research in Sections 5, 6, and 7, respectively.

2. HIGH-LEVEL MODEL

Our recent work formed the basis for the selection of primitives. In particular, in [2] we showed the prominent role of reflection in self-adaptive systems, which informed the general structure of our model, and in [1] we developed a classification of modeling dimensions, which helped us with identifying FORMS's primitives. We then carefully examined each primitive in the context of existing literature (e.g., [12, 13, 16, 10, 9, 8]) to determine its suitability for inclusion in the high-level model. The contribution of the high-level model presented in this section is its ability to precisely represent high-level characteristics of a self-adaptive system, while remaining extensible to enable its specialization for specifying the variation points between the different types of system. In this section we describe and illustrate the high-level model using an intuitive example, and in Section 4 describe its refinement.

2.1 Illustrative Example

We use an unmanned vehicle (UV) system for describing the various concepts in FORMS. As attested to in previous research [4], UVs are shown to be representative of a large class of self-adaptive software systems. A UV needs to be able to autonomously navigate, while taking into account traffic laws and other vehicles. Clearly, the key objective of a UV is to drive to a destination at reasonable speed while avoiding collisions. As further discussed below, a UV is also expected to deal with unanticipated situations that may require adaptation of its software. The software system running on such a platform typically consists of several components, including *control component* used to interface with the mechanical parts, *navigation component* used to determine the best route to the destination, and *awareness component* used to track the location of a car based on the video feed from the camera as well as other sensors. A concrete application of such a system is the DARPA's Grand Challenge contest [17].

A UV system is required to be adaptive to deal with a variety of internal and external changes. For example, due to failure of its camera, a UV system may have to rely on other sensors, such as infrared to temporarily navigate to a safe location. Similarly, depending on the complexity of the terrain, the UV system may choose to select among several alternative navigation components (e.g., computationally expensive but accurate versus efficient but inaccurate).

2.2 The Reference Model

Fig. 1 provides an overview of the FORMS's top-level constructs and their relationships. Though intuitive, the visual representation does not give a precise semantic description of the constructs, which is precisely why throughout this paper we use Z notation for formally specifying the model. 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 [6].

As shown in Fig. 1, a self-adaptive system is situated in an *environment*, and comprises one or more *base-level* and *reflective subsystems*. 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, the environment of a UV includes the road, cars and obstacles on the road, as well as any external software system, such as the GPS system. The distinction between environment and self-adaptive system is made based on the extent of control. For instance, in the UV system, the self-adaptive system has no control over the GPS system (i.e., cannot change it), hence it 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 UV, navigation of the vehicle is performed by base-level 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. For instance, consider a UV that not only has the ability to adapt its navigation strategy (e.g., smooth ride, fastest time, safest), but also adapting the way such adaptation decisions are made (e.g., based on user's preference, gas usage, weather condition).

2.2.1 Environment

As shown in Fig. 1, environment consists of *attributes* and *processes*. An attribute is a perceivable characteristic of the environment. Attributes in a UV may correspond to the location of another car or the current weather condition. The set of attributes is defined:

$[Attribute]$

A process is an activity that can change the environment attributes. For instance, the movement of another UV is considered a process, since it changes the location attribute of that UV. The set of processes is defined:

$$Process == \mathbb{P}\, Attribute \rightarrow \mathbb{P}\, Attribute$$

An environment thus comprises a non-empty set of attributes and a set of processes:

Environment

$$attributes : \mathbb{P}\, Attribute$$
$$processes : \mathbb{P}\, Process$$
$$attributes \neq \varnothing$$

A system's *context* is a set of accessible environment attributes:

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

For example, the context of a UV equipped with a camera includes the location of an observed obstacle on the road, since such environmental attribute can be perceived. On the other hand, the temperature of the road asphalt is clearly an environmental attribute, but not necessarily the context of a UV that has no way of perceiving it. Below we further clarify the meaning of *perceiving* the environment.

2.2.2 Base-Level Subsystem

As depicted in Fig. 1, a base-level subsystem consists of *domain models* and *base-level computations*. However, before we elaborate on these concepts, we need to define the meaning of model and computation in this setting. A model comprises *representations*, which describe something of interest in the physical and/or cyber world:

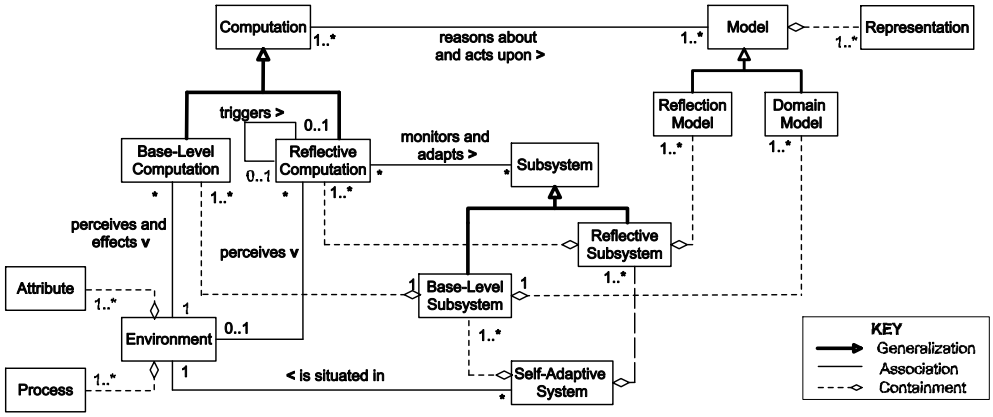

Figure 1: Conceptual view of FORMS's high-level modeling primitives and their relationships.

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

Different models may have different types of representations. An *environment representation* corresponds to an attribute in the environment. The set of environment representations is defined:

$[EnvironmentRepresentation]$

A *domain model* describes (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 UV may incorporate a variety of information: a digital map marked with the roads as well as the location of cars traveling along side the UV on the road, the engine that the base-level software system is controlling, and etc. A domain model maps representations to attribute sets as follows:

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

To define computation, we introduce the type *state*:

$[State]$

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

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

The computation operation computes a new state as follows:

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

For instance, cruise controller computation in the UV would need to maintain its internal state (e.g., accelerate, brake, stop).

We now precisely define base-level computation that is informally depicted in Fig. 1 as follows:

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

A base-level computation reasons and acts upon a subset of domain model by *reading* from and *writing* to it, while at the same time monitors and changes environmental context by *perceiving* and *effecting*. As an example, consider a computation in UV dealing with gas usage that *reads* the current location from the domain model to reevaluate its *state* (e.g., gas usage rate), and based on changes to the state may *write* a new route on the domain model that includes a stop by at a gas station.

A base-level subsystem is a software system that provides the domain functionality (i.e., application logic). Base-level subsystem is defined:

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

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. In the UV example, the base-level subsystem includes the parts of the system dealing with the navigation and control of the vehicle.

2.2.3 Reflective Subsystem

We have previously showed that the key defining characteristic that sets self-adaptive systems apart from traditional computing systems is the ability to reflect on their behavior and structure [2]. Therefore, the modeling primitives dealing with adaptation are informed by concepts that have originated in computational reflection [14]. Fig. 1 shows the two primitives of self-adaptation in FORMS: *reflection model* and *reflective computation*. We specify these primitives below.

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 [14]. A self-adaptive system has a set of reflection model representations:

$[ReflectionModelRepresentation]$

A *reflection model* comprises reflection model representations:

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

Reflection models are used by reflective computations. For instance, a reflection model for the UV may be an architectural representation (e.g., component-and-connector view) of the running software system, which is used at runtime by changing the system's architecture.

A *reflective computation* is defined:

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

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

As mentioned before, a *reflective subsystem* is composed of reflection models and reflective computations. This is formally specified as follows:

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

The specification states that the computations of a reflective subsystem have only access to the models of that subsystem, and they can only trigger computations of the subsystem.

2.2.4 Self-Adaptive System

The formally specified primitives in FORMS provide sufficient expressive power to precisely define a complete self-adaptive system, its relationship with the environment, and most importantly self-adaptation. 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:

$$MetaLevelSubsystem == ReflectiveSubsystem[BaseLevelSubsystem]$$

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

$$MetaMetaLevelSubsystem == ReflectiveSubsystem[MetaLevelSubsystem]$$

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

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

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:

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

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:

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

The specification states that self-adaptation changes the self-adaptive system, but does not effect the environment. The adapta-

tion is performed by one of the meta-level reflective computations (*e?*) which uses its state to adapt one or more base-level subsystems.

3. CASE STUDIES

In this section we apply FORMS to two 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 refine and specialize the key primitives. We start by applying the reference model to IBM's MAPE-K model [12]. Then we study a concrete approach for self-adaptation in the robotics domain [9].

3.1 MAPE-K

As a first case, we study the MAPE-K reference model [12] and the corresponding reference architecture, which advocates a hierarchical composition of autonomic systems. Fig. 2 shows an autonomic manager, the basic building block of an autonomic system. The autonomic control loop consists of four basic activities: monitor, analyze, plan, and execute.

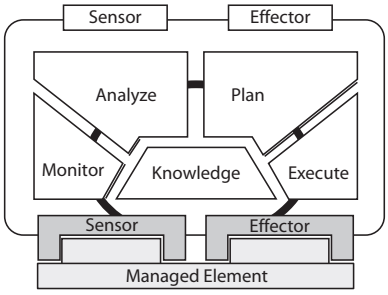

Figure 2: MAPE-K [12]

The activities share knowledge that typically includes a model of the managed element and a description of goals [11]. Monitor uses sensors to collect data from the managed element, which could be either a software/hardware resource, or an autonomic manager itself. Analyze provides mechanisms for interpreting the collected data and predicting the future situations. Plan constructs the actions needed to achieve goals. Finally, execute uses effectors to carry out changes to managed elements. An autonomic manager provides sensors and effectors for other autonomic managers to use. As further detailed below, this enables hierarchical composition of autonomic managers.

Fig. 3 shows the MAPE-K instantiation of the basic concepts of FORMS. In MAPE-K, an *autonomic manager* corresponds to a *reflective subsystem* from FORMS. An *autonomic manager* comprises four types of *autonomic manager components*, 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 *reflective subsystem*). 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 MAPE-K, the role of *knowledge* is less clear. According to Kephart and Chess [12], *knowledge* refers to the data collected from *managed resources*, models for analysis such as queueing network models, policy information, and action plans. Miller [15] groups the different forms of knowledge in three distinct types: *topology knowledge*, *policy knowledge*, and *problem determination knowledge*. Huebscher and McCann [11] elaborate on knowledge and make a distinction between: (1) architectural model of the managed element, including structural and behavioral data; (2) goal model (e.g. event-condition-action policies or utility functions); and (3) model of the operating environment in which the managed element is situated (e.g., hardware devices, network connections), which may also include other autonomic elements.

We now briefly explain how hierarchies of autonomic systems are constructed using *autonomic managers*. Fig. 4 shows the different types of autonomic managers proposed in [12].

A *resource autonomic manager* manages a *managed resource*. Four concrete types are distinguished: managers for *self-configuring*, *self-optimizing*, *self-healing*, and *self-protecting*. *Orchestrating autonomic managers* on the other hand manage groups of *resource autonomic managers*. In particular, an *orchestrating autonomic manager within discipline* manages a group of *resource autonomic managers* of the same type, while an *orchestrating autonomic manager across disciplines* 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 *reflective subsystem* in FORMS that can be reflected upon by a *reflective subsystem* from the level above. A hierarchy of *autonomic managers* in MAPE-K 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 [15], a *knowledge source* is an implementation of a registry, dictionary, database or other repository that provides access to knowledge that need to be shared among autonomic managers. As further discussed in Section 4, this is an area of further refinement in FORMS.

3.2 Robotics

In our second case, we study a system from the robotics domain. Edwards et al. [9] propose an approach to design and implementation self-adaptive behavior in robotics software through layers. In particular, they propose an adaptive robotic software architecture consisting of (1) a basic bottommost layer with application components that control the robot, and (2) one or more meta-layers with meta-level components that implement fault-tolerance, dynamic update, resource discovery, redeployment, and etc. In the proposed architecture, each layer may adapt the layer beneath. Fig. 5 shows one application instance [9], where component instances are layered and distributed over several nodes. The *robot behavior* (bottommost layer) provides the robot's application logic. In this case the system is distributed on two robots (nodes), where a robot follower trails the leader. On top of that, using meta-level components, we find 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 our high-level FORMS model we pinpointed two pivotal concepts for self-adaptive systems: *reflective computations* and *reflective models*. We now study these concepts in more depth for this specific case. Fig. 6 shows the relationship between the entities in the concrete robotics system of Fig. 5 and FORM's high-level primitives.

In the first case study we identified four types of *reflective com-*

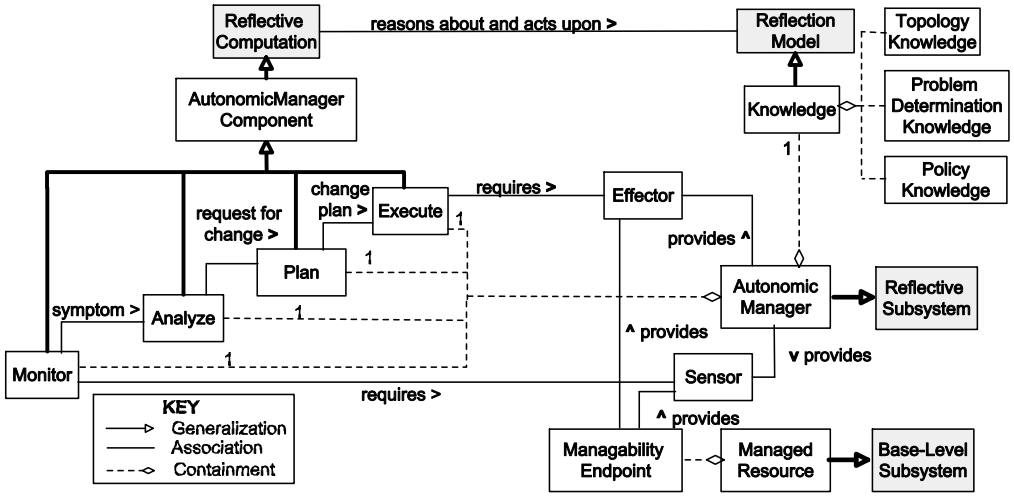

Figure 3: Conceptual view of MAPE-K's computations and knowledge in relation to FORMS primitives (shades boxes).

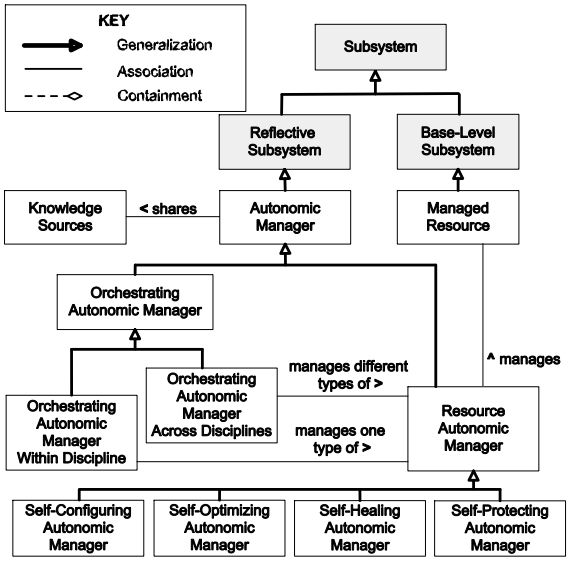

Figure 4: Hierarchies of autonomic managers in relation to FORMS.

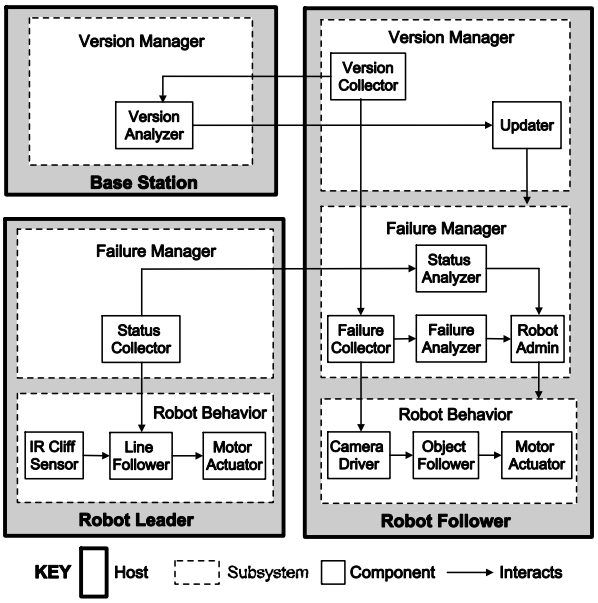

Figure 5: Overview of the robotics architecture presented in [9].

putations that corresponded to the four MAPE stages: monitor, analyze, plan, and execute. In this case study, we see a similar set of computations represented by the *meta-level components*. A *meta-level component* corresponds to the FORMS's *reflective computation*. The authors propose three concrete *meta-level components*: *collector*, *analyzer*, and *admin*. *Collector* gathers data from the components at the layer below. The collected data could be both behavioral, such as the utilization of resources, and structural, such as the topology (configuration) of component instances. *Analyzer* interprets the data produced by *Collectors* to evaluate adaptation policies and create adaptation plans. *Admin* controls and manipulates components at the layer below according to the adaptation plans produced by *analyzers*. A difference compared to MAPE-K is that in this architecture analysis and planning are merged into the *analyzer* component.

Edwards et al. describe different types of data and models used by the *meta-level components*. These models correspond to the notion of *reflection model* in FORMS. The different types of models used in this case study helped us to further refine FORMS. The architecture makes use of four types of models: (1) All *meta-level components* within a layer have access to a *runtime system architecture* that contains the constructs (components, connectors, ports, etc.) of the layer below. The components of the meta-layer that deal with *failure manager* layer have access to the *runtime system architecture* of the *robot behavior* layer, which corresponds to the *base-level subsystem* in FORMS. *runtime system architecture* resembles *topology knowledge* in MAPE-K. (2) *Collector* computations produce *collector data* that *analyzer* computations consume. For example, the *failure collector* monitors the *camera driver* and reports failures to *failure analyzer*. *Collector data* is similar to *problem determination knowledge* in MAPE-K. (3) A layer is concerned with a specific concern, which is represented as an *adaptation policy*. For example the *failure manager* layer is concerned

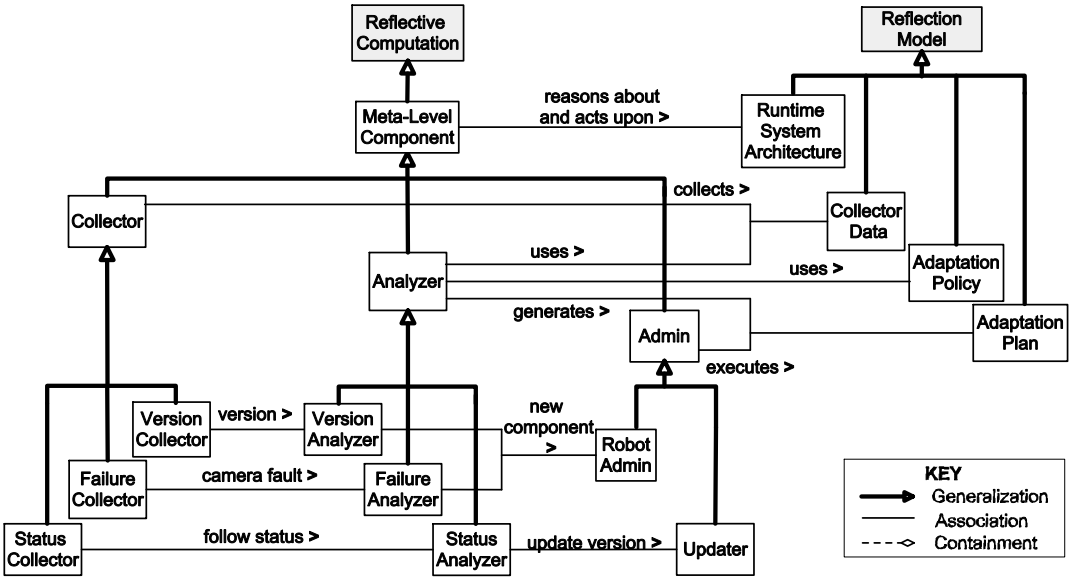

Figure 6: Conceptual view of the robotics system's computations in relation to FORMS.

with reactive fault recovery. *Adaptation policy* is similar to *policy knowledge* in MAPE-K. *Analyzer* computations use *adaptation policies* to deduce a suitable *adaptation plan*. The *failure analyzer* in the example determines the best replacement component for the camera based on *adaptation policies*. (4) The *adaptation plan* is consumed by *admin* computations. For instance, the *failure analyzer* in the example notifies the *robot admin* of the new component that is needed, and the *robot admin* instantiates the component.

As detailed in the next section, the different type of models (knowledge) identified in this case study helped us with refining FORMS's *reflection model* primitive.

4. ELABORATED MODEL

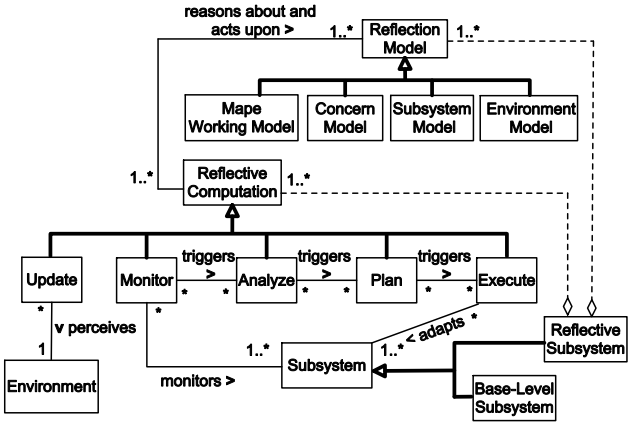

Figure 7: Refined view of FORM's primitives and their relationships.

Using FORMS's high-level primitives to analyze the two case studies, as well as several other systems that for brevity are not reported here, helped us to identify a number of additional extensions and refinements. This was anticipated, since as you may recall from Section 2 we had intentionally selected to include only the minimum set of primitives required for formally specifying self-adaptation. Moreover, this was desired, since our objective has been to discern the elements of FORMS by carefully exploring the existing systems, as opposed to simply devising them through hypothetical means. It is also only through further refinement of the FORMS's high-level primitives that we can achieve the required level of detail for distinguishing between different types of self-adaptive systems. Fig. 7 shows the elaborated model of FORMS, including the additional primitives and their relationship to the high-level constructs. Below we formally describe the detailed model.

Reflection Models. We distinguish between four types of reflection models: *subsystem model*, *concern model*, *environment model*, and *mape working model*. A *subsystem model* provides an abstract representation of the managed subsystem, often in the the form of an architectural model used at runtime, such as *topology knowledge* in MAPE-K and the *runtime system architecture* from the robotics case study. A *concern model* represents the reflective subsystem's objectives (i.e., goals). As you may recall from the case studies, in both MAPE-K and robotics example, each layer is concerned with a specific set of objectives. In the robotics case study, the *version manager* is concerned with updating the robots, while the *failure manager* is concerned with the system failures. The *environment model* represents the contextual information utilized in making the decisions in the meta layers. For instance, in the robotics example, the meta-meta-layer dealing with version management updates the system's failure management components only when the robot is at a base station. In other words, the location of the robot is an environmental attribute used by the reflective subsystem for making the adaptation decisions. Finally, a *mape working model* represents the data structures and information shared between the MAPE activities. For instance, as you may recall from Section 3, when the *plan* component generates a new *solution*, it provides that via a *mape working model* for later use by the *execute* component.

We now formally specify the environment and subsystem models. Due to space limitations, the specification of the other models is provided at [20].

An environment model comprises representations of attributes in

the environment relevant for a particular concern of interest. Environment models are defined:

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

The predicates state that the environment representations defined in reflection model map to sets of attribute of the environment.

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

$$[\mathit{Feature}]$$

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

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

We introduce subsystem representation that represents (a part of) a subsystem, which can be either a base-level subsystem or a reflective subsystem. Subsystem representations are defined:

$$
\begin{array}{|l}
\hline
\mathit{SubsystemRepresentation}\ [\mathit{Subsystem}] \\
\hline
\\
\hline
\end{array}
$$

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

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

The generic schema states that subsystems representations defined in reflection model map to sets of features of that subsystem (returned by the reify function). We can apply the generic scheme to define models for concrete types of subsystems. For example, a base-level subsystem model is defined:

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

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

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

Reflective Computations. We define five types of reflective computations for self-adaptive system: *update*, *monitor*, *analyze*, *plan* and *execute*. The last four represent the four MAPE activities, whose relationships to FORMS became clear through application of FORMS to both the MAPE-K and the robotics case studies. *Update* represents an additional reflective computation that provides the linkage between the environment and reflective subsystem. For instance, as you may recall from Section 3.2, the location of the robot is an environmental attribute required by *version manager* for determining when the adaptation should occur. In FORMS we make an explicit distinction between the process that maintains the *subsystem model* (*monitor*) and the process that maintains the *environment model* (*update*). Note that reflective computations can only act upon the reflection models of the reflective subsystem to which the computations belong. However, this does not exclude that reflective subsystems may share reflection models. An example of sharing of models are *knowledge sources* in MAPE-K.

In this paper, we present the specification of *execute*. The other computations are specified similarly. For space constraints, we have provided them at [20].

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

Execute computations can 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. Due to space constraints, we present the specification of executing, and provide the additional details in [20].

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

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

We introduce the schema *reflective computations* which groups the computations of a reflective subsystem:

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

The specification states that monitor computations can trigger analyse computations, analyse computations can trigger plan com-

putations, and plan computations can trigger execute computations. This loop of triggers corresponds to the autonomic loop in MAPE-K and the sequence of events between meta-level components in the robotics case study.

Examples. We conclude the formal specification with modeling two examples from the case studies using FORMS. First, we specify an autonomic system consisting of a two-level hierarchy of *autonomic managers*. Then, we specify the architecture of the robotics case with two robots.

The first example consists of a self-adaptive autonomic system with four resources: a server, two clients, and a network.

SelfAdaptiveAutonomicSystem

$resources : \mathbb{P}\ ManagedElement$
$resourceManagers : \mathbb{P}\ ResourceAutonomicManager$
$systemManager :$
$\quad \mathbb{P}\ OrchestratingAutonomicManagerWithinDiscipline$
$serverExecute, networkExecute, systemExecute : Execute$

$resources = \{server, client1, client2, network\}$
$resourceManagers = \{serverOptimizer, networkOptimizer\}$
$systemManager = \{systemOptimizer\}$
$serverExecute \in serverOptimizer.computations$
$networkExecute \in networkOptimizer.computations$
$systemExecute \in systemOptimizer.computations$
dom $serverExecute.adapt =$
$\quad \{(serverExecute.knowledge, \{server\})\}$
dom $networkExecute.adapt =$
$\quad \{(networkExecute.knowledge, \{network\})\}$
dom $systemExecute.adapt = \{(systemExecute.knowledge,$
$\quad \{serverExecute, networkExecute\})\}$

One *resource manager* is managing the *server*, another one is managing the *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 of the autonomic managers in the self-adaptive autonomic system.

The second example describes a self-adaptive robotic system that corresponds to the structure shown in Fig. 5.

The *base-level systems* in this example are *robot behaviors*.

RobotBehavior

$BaseLevelSubsystem$

The *reflective subsystems* are the *managers*. Managers have three types of computations: *collectors*, *analyzers*, and *admins*. We show the definition of *collector*; the other models are provided at [20].

Collector $[Subsystem]$

$Monitor[Subsystem]$
$c\_trigger : \mathbb{P}\ State \times \mathbb{P}\ Analyse2Plan[Subsystem] \rightarrow$
$\quad \mathbb{P}\ Analyse2Plan[Subsystem]$

Managers use different types of models: *environment models* (i.e., location of robots), *runtime system architecture*, *collector data*, *adaptation policies*, and *adaptation plans*. We show the definition of *collector data* model:

CollectorData

$MapeWorkingModel$

In addition, we illustrate the concrete *environment model* of a version manager:

$versionEnvironmentModel : EnvironmentModel$

$representations = \{followerPosition, baseStationLocation\}$

Version manager is defined:

VersionManager

$ReflectiveRoboticSubsystem[FailureManager]$

$environmentModels = \{versionEnvironmentModel\}$
$concernModels = \{versionAdaptationPolicy\}$
$mapeWorkingModels = \{failureAdaptationPlan\}$
$subsystemModels = \{failureArchitecture\}$
$collectors = \{versionCollector\}$
$analyse2Planners = \{versionAnalyzer\}$
$executes = \{updater\}$

The *version manager* uses the *position* of the *follower* and the *location* of the *base station* in the *environment model* to determine whether the robot has reached the base station.

Finally, a *self-adaptive robotic system* is defined:

SelfAdaptiveRoboticSystem

$robotBehaviors : \mathbb{P}\ RobotBehavior$
$failureManagers : \mathbb{P}\ FailureManager$
$versionManagers : \mathbb{P}\ VersionManager$

$robotBehaviors = \{leaderBehavior, followerBehavior\}$
$failureManagers = \{failureManager\}$
$versionManagers = \{versionManager\}$
$\ldots$
dom $versionCollector.sense =$
$\quad \{(\{failureManager\}, versionCollector.state)\}$
dom $versionCollector.c\_trigger =$
$\quad \{(versionCollector.state, \{versionAnalyzer\})\}$
dom $versionAnalyzer.ap\_trigger =$
$\quad \{(versionAnalyzer.state, \{updater\})\}$
dom $updater.adapt = \{(\{failureManager\}, updater.state)\}$
$\ldots$

We only show the part of the specification that defines the architectural structure of the *version manager* and the scope of the *sense* computation (i.e. the *failure manager*), the interactions between the computations (*triggers*) and the scope of the *adaptation* computation (also the *failure manager*). The complete specification is provided at [20].

5. RELATED WORK

The literature on self-adaptive and autonomic software systems is extensive. Two of the main influences on the work presented herein are computational reflection and the use of control loops in software design. In her seminal work [14], Maes stresses the role of computational reflection in programming software systems that are capable of introspecting their runtime properties, and accordingly adapt their behavior. Coulson et al. [5] describe an implementation model for reflective middleware. Early work of Shaw [18] presents an approach for self-adaptation based on process control loops. Dobson et al. [7] and Brun et al. [3] promote feedback control-loop as an important design concept in engineering of self-adaptive software systems. In Fig. 1 one sees clearly how these concepts influence FORMS. A problem is the lack of formal description that provides for the required precision and expressibility.

FORMS also builds extensively on the previous frameworks and reference implementations for self adaptive and autonomic systems [12, 13, 16, 10, 9, 8]. We have comprehensively discussed some models and their contributions to FORMS earlier. We have also shown how, as opposed to these models, FORMS takes the first steps towards a precise, formally specified, reference model.

Formal models targeting specific aspects of self-adaptation exist. For instance, Zhang et al. [21] presents an approach to formally model the behavior of adaptive programs, automatically analyze them, and generate an implementation of the system. Wermelinger and Fiadeiro [19] present an algebra for formally specifying runtime reconfiguration of a system's software architecture. These and

other works demonstrate the usefulness of applying formal modeling to this field. Nevertheless, none of these works have intended to provide a generally applicable reference model for expressing, comparing, and evaluating different types of self-adaptive software.

6. APPLICATIONS OF FORMS

Unlike existing informal autonomic frameworks that at best provide the engineers with a set of rough guidelines, we have used a formal language for rigorous specification of our reference model. This formalism affords numerous capabilities. For instance, existing tools could be used for: (1) *type checking* (e.g., CZT's type checker [6]) to automatically obtain certain guarantees on the validity of the design of a self-adaptive system, such as conformance of architecture models that are refined iteratively, (2) *executing* and *animating* the schemas (e.g., CZT's ZLive animator [6]) to visually obtain a better understanding of the system's properties, (3) *testing* (e.g., CZT's ModelJUnit [6]) to automatically generate test cases for the self-adaptive system, and (4) *model transformation* to automatically transfer FORMS to other formal notations for performing various types of analysis (e.g., concurrency checks) or even generate skeleton code that provides partial implementation of the system. The latter would be in particular useful for ensuring the fidelity of the implemented system to its specification in FORMS.

Beyond automated tool support, formal specification has been shown to increase the quality of software development, as formal reasoning can help to detect problems at early stages of system development. Moreover, our approach allows validation of a system built according to its specification in FORMS against the requirements, i.e., translate a specific requirement into a predicate and determine whether it follows from the system's specification.

In light of the above discussion, the contributions of FORMS can be summarized as follows: (1) establishes a shared vocabulary of primitives in this domain that while simple and concise can be used to precisely define arbitrary complex self-adaptive systems, (2) enables engineers to precisely express their design choices and assess those choices using the mechanisms outlined above, (3) allows for comparison and evaluation of different types of self-adaptive systems, and (4) lays the foundation for a systematic method of developing a catalog of known solutions (i.e., architectural patterns).

7. CONCLUSIONS & FUTURE WORK

In this paper, we presented a novel formal reference model for self-adaption, entitled FORMS. We have distilled self-adaptation primitives from existing frameworks and established principles, related them to one another in an integrated conceptual model, and provided a formal definition of them using Z notation. We have demonstrated FORMS precision, expressiveness, and extensibility by applying it to several systems.

While our experiences with FORMS have been very positive, several avenues of future work remain. We plan to study other types of systems, such as those that are biologically inspired, to further assess, and potentially extend, FORMS's primitives. For instance, we envision the need for refining the existing primitives to represent coordination in decentralized systems. 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 specific quality attributes, we intend to develop a catalog of strategies and tactics for building systems in this domain. In the long-term, FORMS may be the foundation for a full-fledged model based engineering approach for self-adaptive and autonomic software systems.

8. REFERENCES

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

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

[3] Y. Brun et al. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems*. LNCS Hot Topics, 2009.

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

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

[6] CZT. `http://czt.sourceforge.net/`, Jan 2010.

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

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

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

[10] D. Garlan et al. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, Oct 2004.

[11] M. C. Huebscher and J. A. McCann. A survey of autonomic computing - degrees, models, and applications. *ACM Comput. Surv*, 40(3), 2008.

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

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

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

[15] B. Miller. The autonomic computing edge: The role of knowledge in autonomic systems., Sep 2005.

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

[17] G. Seetharaman et al. Unmanned vehicles come of age: The DARPA grand challenge. *IEEE Computer*, Dec. 2006.

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

[19] M. Wermelinger and J. L. Fiadeiro. Algebraic software architecture reconfiguration. In *European Software Engineering Conf. and Int'l Symp. on Foundations of Software Engineering*, Toulouse, France, 1999 1999.

[20] D. Weyns, S. Malek, and J. Andersson. Z specifications of FORMS. In *CW 579, Tech. Report, Katholieke Universiteit Leuven*, 2010. `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW579.abs.html`.

[21] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *International Conference on Software Engineering*, Shanghai, China, May 2006.