Autonomous Distributed Control and Management via MAPE

Bachelor Thesis

durchgeführt am
Lehrstuhl für Netzarchitekturen und Netzdienste
Fakultät für Informatik
Technische Universität München

von
cand. Bachelor inform.

Enrique Abdón García Pérez

Technische Universität München
Fakultät für Informatik
Lehrstuhl für Netzarchitekturen und Netzdienste

Autonomous Distributed Control and Management via MAPE

–

Autonomous Distributed Control and Management via MAPE

Bachelor Thesis

durchgeführt am
Lehrstuhl für Netzarchitekturen und Netzdienste
Fakultät für Informatik
Technische Universität München

von
cand. Bachelor inform.

Enrique Abdón García Pérez

Aufgabensteller:	Prof. Dr.-Ing. Georg Carle
Betreuer:	Dipl.-Inform. Marc Oliver Pahl
Zeitraum :	von 01.September 2009 - bis 01. März 2010
Tag der Abgabe:	01. März 2010

Motivation:

Current research in autonomic computing suffers from the lack of a common definition of the basic autonomic entities. Defining and developing the basic autonomic entities would greatly simplify prototyping of different self-management techniques and permit autonomic techniques to be compared. Although different aspects of autonomic computing are explored in isolation, the control cycle of an autonomic element itself has not been completely defined.

This thesis presents a design of an autonomic element in a distributed object environment. The goal of this architectural design is to provide an easy to understand and modify autonomic element which can be used in most domains with only minor modifications.

Contents

1. Introduction

A computer system usually starts as a simple clean system intended for a well defined environment and applications. However, in order to deal with growth and new demands, storage, computing and networking components are added, replaced and removed from the system, while new applications are upgraded.

The autonomic computing effort aims to make systems self-configuring and self-managing. In order to accomplish this goal, all system components need to be perform autonomic functionalities themselves. Each such component has its own policy for how to react to change in its environment.

The increasing complexity of computing systems is overwhelming the capabilities of software developers and system administrators who design, evaluate, integrate, and manage these systems. Today, computing systems include very complex infrastructures and operate in complex heterogeneous environments. With the proliferation of handheld devices, the expanding number of users, computing vendors have difficulty providing an infrastructure to address all the needs of users, devices, and applications.

SOAs with Web services as their core technology have solved many problems, but they have also raised numerous complexity issues. One approach to deal with the business challenges arising from these complexity problems is to make the systems self-managed or autonomic. For a typical information system consisting of an application server, a Web server, messaging facilities, and layers of middleware and operating systems, the number of variables to configure exceeds, in some cases, human comprehension and analytical capabilities. Major software and system vendors try to create autonomic systems by developing methods, architecture models, middleware, algorithms, and policies to mitigate the complexity problem.

By attacking the software complexity problem through technology simplification and automation, autonomic computing also tries to solve some software evolution problems. Instrumenting software systems with autonomic technology will allow us to monitor or verify requirements (functional or nonfunctional) over long periods of time. For example, self-managing systems will be able to monitor and control the brittleness of legacy systems, provide automatic updates to evolve installed software, adapt safetycritical systems without halting them, immunize computers against malware automatically, facilitate enterprise integration with self-managing integration mechanisms, document architectural drift by equipping systems with architecture analysis frameworks, and keep the values of quality attributes within desired ranges.

In order to accomplish all the above functionalities, autonomic computing needs the *autonomic element* which is the fundamental building block of any autonomic system. This thesis main focus is the design of an autonomic element. Chapter 2 briefly presents the basic concepts of autonomic computing, pointing the importance of the autonomic element in autonomic networks. In chapter 3 we introduce ACMP as the platform for which the autonomic element is designed. This doesn't mean that our design is not extensible for other autonomic frameworks, but that the proof of concept is for ACMP. In chapter 4 we present and evaluate other research works with similar purpose as ours.

The above chapters can be considered as the introduction, in chapter 5 we present specific goals and scope of this thesis. The architecture of the proposed *autonomic element* is presented in chapter 6. As mentioned before this prototype was implemented for ACMP. In order evaluate our design we include chapter 7. Last we present our conclusion and outlook in chapter 8.

2. Concepts and Foundation of Autonomic Computing

This thesis focuses on the design of an autonomic element, which is a fundamental component of an autonomic system. In this chapter we explain the basic concepts of autonomic computing. We also introduce the concept of control theory since the autonomic element is presented as a control system. Last we describe policy management as a higher abstraction layer to define goals for the autonomic element.

According to the autonomic computing paradigm, computing systems possess the properties and capabilities of self-awareness and self-management. An autonomic computing system can be defined as an "Intelligent" open system that [**?**, **?**, **?**, **?**]:

- "knows" itself.
- manages task complexity.
- continuously tunes itself
- prevents and recovers from failures
- provides a safe environment

In this section we will describe the basic concepts needed to understand how the autonomic computing goals are achieved.

2.1 The Control Loop

At the heart of an autonomic system is a control system, which is a combination of components that act together to maintain actual system attribute values close to desired specifications. Control theory [**?**] deals with influencing the behavior of dyanamical systems such as autonomic systems. As an example of the application of control theory, consider an automobile's cruise control, which is a device designed to mantain a contant speed; the *desired* speed, provided bye the driver . The system in this case is the vehicle. The system output is the vehicle speed,the system input is the desired speed, and the control variable is the engine throttle position which inlfuences the engine torque output.

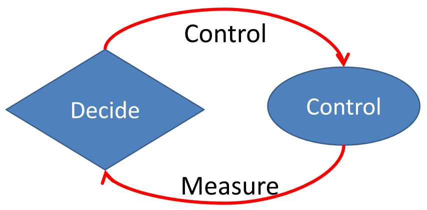

Figure 2.1: Closed control loop.

A primitive way to implement cruise control is simply to lock the throttle position when the driver engages cruise control. However, on mountain terrain, the vehicle will slow down going uphill and accelerate going downhill. In fact, any parameter different than what was assumed at design time will translate into a proportional error in the output velocity, including exact mass of the vehicle, wind resistance, and tire pressure. This type of controller is called an *open-loop* controller because there is no direct connection between the output of the system (the vehicle's speed) and the actual conditions encountered; that is to say, the system does not and can not compensate for unexpected forces.

In a *closed-loop* control system, a sensor monitors the output (the vehicle's speed) and feeds the data to a computer which continuously adjusts the control input (the throttle) as necessary to keep the control error to a minimum (that is, to maintain the desired speed). Feedback on how the system is actually performing allows the controller (vehicle's on board computer) to dynamically compensate for disturbances to the system, such as changes in slope of the ground or wind speed. An ideal feedback control system cancels out all errors, effectively checking the effects of any forces that might or might not arise during operation and producing a response in the system that perfectly matches the user's wishes. In reality, this cannot be achieved due to measurement errors in the sensors, delays in the controller, and imperfections in the control input.

An autonomic system embodies one or more closed control loops. A closed-loop system includes some way to sense changes in the managed element, so corrective action can be taken. The speed with which a simple closed-loop control system moves to correct its output is described by its damping ratio and natural frequency[**?**]. Properties of a control system include spatial and temporal separability of the controller from the controlled element, evolvability of the controller, and filtering of the controlled resource.

Numerous engineering products embody open-loop or closed-loop control systems. The Autonomic Computing(AC) community often refers to the human autonomic nervous system (ANS) [**?**] with its many control loops as a prototypical example. The ANS monitors and regulates vital signs such as body temperature, heart rate, blood pressure, pupil dilation, digestion blood sugar, breathing rate, immune response, and many more involuntary, reflexive responses in our bodies[**?**]. The ANS consists of two separate divisions called the parasympathetic nervous system, which regulates day-to-day internal processes and behaviors, and the sympathetic nervous system, which deals with stressful situations[**?**]. Studying the ANS might be instructive for the design of autonomic software systems. For example, physically separating the control loops that deal with normal and abnormal situations might be a useful design idea for autonomic software systems.

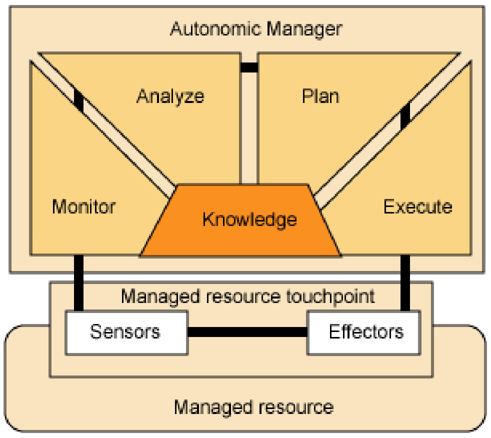

Figure 2.2: Autonomic Element-Basic Architecture [?].

2.2 Autonomic Element

IBM researchers have established an architectural framework for autonomic systems [?]. An autonomic system consists of a set of autonomic elements that contain and manage resources and deliver services to humans or other autonomic elements. An autonomic element consists of one autonomic manager and one or more managed elements. The core of an autonomic element is a control loop 2.1 that integrates the manager with the managed element. The autonomic manager consists of sensors, effectors, and a five-component analysis and planning engine as depicted in Figure 2.2. The monitor observes the sensors, filters the data collected from them, and then passes the distilled data to the analysis engine. The analysis engine compares the collected data against the desired sensor values also stored in the knowledge base. The planning engine devises strategies to correct the trends identified by the planning engine. The execution engine finally adjusts parameters of the managed element by means of effectors and stores the affected values in the knowledge base.

An autonomic element manages its own internal state and its interactions with the environment. An element's internal behavior is driven by the goals and policies the designers have built into the system.

Autonomic elements can be arranged as strict hierarchies or graphs. Touch points represent the interface between the autonomic manager and the managed element. Through touch points, autonomic managers control a managed resource or another autonomic element. It is imperative that touch points are standardized, so autonomic managers can manipulate other autonomic elements in a uniform manner. That is, a single standard manageability interface, as provided by a touch point, can be used to manage routers, servers, application software, middleware, a Web service, or any other autonomic element. This is one of the key values of AC: a single manageability interface, rather than the numerous sorts of manageability interfaces that exist today, to manage various types of resources [?]. Thus, a touch point constitutes a level of indirection and is the key to adaptability.

A manageability interface consists of a sensor and an effector interface. The sensor interface enables an autonomic manager to retrieve information from the managed element through the touchpoint using two interaction styles: (1) request-response for solicited (queried) data retrieval and (2) send-notification for unsolicited (event-driven) data retrieval. The effector interface enables an autonomic manager to manage the managed element through the touchpoint with two interaction types: (1) perform-operation to control the behavior (e.g., adjust parameters or send commands) and (2) solicit-response to enable call-back functions.

2.3 Characteristics of Autonomic Systems

An autonomic system can self-configure at runtime to meet changing operating environments, self-tune to optimize its performance, self-heal when it encounters unexpected obstacles during its operation, and protect itself from malicious attacks. Research and development teams concentrate on developing theories, methods, tools, and technology for building self-healing, self-configuring, self-optimizing, and self-protecting systems.

An autonomic system, as described in [?, ?], has the following characteristics:

- self-configuring: Self-configuring systems provide increased responsiveness by adapting to a dynamically changing environment. A self-configuring system must be able to configure and reconfigure itself under varying conditions. Different degrees of end-user involvement should be allowed, from user-based reconfiguration to automatic reconfiguration based on monitoring and feedback loops[?]. For example, the user may be given the option of reconfiguring the system at runtime; alternatively, adaptive algorithms could learn the best configurations to achieve mandated performance or to service any other desired functional or nonfunctional requirement.

- self-optimizing: Self-optimizing systems provide operational efficiency by balancing resources and workloads. Such a system will continually monitor and balance its resources and operations. In general, the system will continually seek to optimize its operation with respect to a set of prioritized requirements to meet the needs of the application environment. Capabilities such as repartitioning, reclustering, load balancing, and rerouting must be designed into the system to provide self-optimization. Adaptive algorithms, along with other systems, are needed for monitoring and response.

- self-healing: Self-healing systems provide resiliency by discovering and preventing disruptions as well as recovering from malfunctions. Such a system will be able to recover -without loss of data or noticeable delays in processing - from routine and extraordinary events that might cause some of its parts to malfunction. Self-recovery means that the system will select, possibly with user input, an alternative configuration to the one it is currently using and will switch to that configuration with minimal loss of information or delay.

- self-protecting: Self-protecting systems secure information and resources by anticipating, detecting, and protecting against attacks. Such a system will be capable of protecting itself by detecting and counteracting threats through the use of pattern recognition and other techniques. This capability means that the design of the system will include an analysis of the vulnerabilities and the inclusion of protective mechanisms that might be employed when a threat is detected. The design must provide for capabilities to recognize and handle different kinds of threats in various contexts more easily.

- reflexivity: An autonomic system must have detailed knowledge of its components, current status, capabilities, limits, boundaries, interdependencies with other systems, and available resources. Moreover, the system must be aware of its possible configurations and how they affect particular nonfunctional requirements.

- adapting: Most of the characteristics listed above are founded on the ability of an autonomic system to monitor its performance and its environment and respond to changes by switching to a different behavior. At the core of this ability is a control loop. Sensors observe an activity of a controlled process, a controller component decides what has to be done, and then the controller component executes the required operations through a set of actuators. The adaptive mechanisms to be explored will be inspired by work on machine learning, multi-agent systems, and control theory.

2.4 Policies

Autonomic elements can function at different levels of abstraction. At the lowest levels, the capabilities and the interaction range of an autonomic element are limited and hard-coded. At higher levels, elements pursue more flexible goals specified with policies, and the relationships among elements are flexible and may evolve. Kephart and Walsh proposed a unified framework for AC policies based on the well-understood notions of states and actions [**?**]. In this framework, a policy will directly or indirectly cause an action to be taken that transitions the system into a new state. Kephart and Walsh distinguish three types of AC policies, which correspond to different levels of abstraction, as follows:

1. Event Condition Action(ECA) policies: An ECA policy dictates the action that should be taken when the system is in a given current state. Typically this action takes the form of "IF (condition) THEN (action)" where the condition specifies either a specific state or a set of possible states that all satisfy the given condition. The state that will be reached by taking the given action is not specified explicitly. Presumably, the author knows which state will be reached upon taking the recommended action and deems this state more desirable than states that would be reached via alternative actions. This type of policy is generally necessary to ensure that the system is exhibiting rational behavior.

2. goal policies: Rather than specifying exactly what to do in the current state, goal policies specify either a single desired state, or one or more criteria that characterize an entire set of desired states. Implicitly, any member of this set is equally acceptable. Rather than relying on a human to explicitly encode rational behavior, as in action policies, the system generates rational behavior itself from the goal policy. This type of policy permits greater flexibility and frees human policy makers from the "need to know" low-level details of system function, at the cost of requiring reasonably sophisticated planning or modeling algorithms.

3. utility-function policies: A utility-function policy is an objective function that expresses the value of each possible state. Utility-function policies generalize goal policies. Instead of performing a binary classification into desirable versus undesirable states, they ascribe a real-valued scalar desirability to each state. Because the most desired state is not specified in advance, it is computed on a recurrent basis by selecting the state that has the highest utility from the present collection of feasible states. Utility-function policies provide more fine-grained and flexible specification of behavior than goal and action policies. In situations in which multiple goal policies would conflict (i.e., they could not be simultaneously achieved), utility-function policies allow for unambiguous, rational decision making by specifying the appropriate tradeoff. On the other hand, utility-function policies can require policy authors to specify a multidimensional set of preferences, which may be difficult to elicit; furthermore they require the use of modeling, optimization, and possibly other algorithms.

2.5 Autonomic Networking

This section will examine autonomic networking, which is a new form of autonomic computing that was developed to specifically address the needs of embedding increased intelligence and decision making in networks, network devices and networked applications.

The name 'autonomic' was deliberately chosen by IBM to invite comparisons to biological mechanisms. The vast majority of past and current work on autonomic computing focuses on IT issues, such as host systems, storage and database query performance. However, networks and networked devices and applications have different needs and operate under different assumptions than IT issues. Autonomic networking focuses on the application of autonomic principles to make networks and networked devices and applications more intelligent, primarily by enabling them to make decisions without having to consult with a human administrator or user.

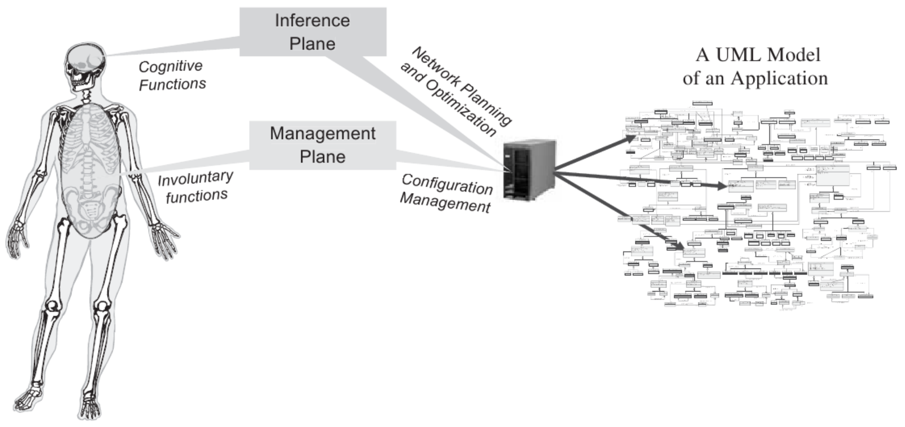

Figure 2.3: Analogies between human body and autonomic systems.

The motivation behind autonomic networking is simple: identify those functions that can be done without human intervention to reduce the dependence on skilled resources. This not only lowers operational costs, but also enables these valuable resources to be used for more complex tasks. It also enables the network to respond faster and more accurately to the changing needs of users and businesses. The left of Figure 2.3 shows a human body, while the picture on the right is a UML model of a simple application. The autonomic nervous system and the brain in the human body are the 'management plane' and 'inference plane', respectively, of autonomic networking, which are explained below.

The first, and most important analogy, is that complexity is everywhere. Hence, autonomics is, first and foremost, a way to manage complexity. This is conceptualized by the complexity of the UML application shown in figure 2.3 - the number of components and their complicated interconnection approaches that of the complexity of the interconnection of the autonomic nervous system with components of the surrounding human body. The networking analogy to this is as follows: the involuntary functions of the body equate to manual, time-consuming tasks (such as configuration management) performed by the network administrator, while the cognitive functions of the brain compare to higher level, more complex system-wide functions, such as planning and optimization of the network. In other words, autonomics does not necessarily replace humans; rather, it performs time-consuming tasks so that skilled human resources can be better leveraged.

This idea - automating simple, safe decisions and corrective actions - can be extrapolated into a new management approach, one that will be particularly useful for next generation networks as well as cognitive networks[**?**]. This new management approach is founded on the realization that multiple networks as well as networks with different functionalities will use multiple control mechanisms that are not compatible with each other. Figure 2.3 shows a conceptual view of how networks are traditionally managed. In this approach, the data plane is a contiguous path used for data communications, while the control plane is used to set up, manage and terminate connections in the data plane.

2.5.1 Managing Complexity

One of the difficulties in dealing with complexity is accommodating legacy hardware and software. Current approaches to network configuration and management are technology-specific and network-centric, and do not take business needs into account. For example, monitoring and (re)configuration are performed on most network devices using one or more management protocols and languages. The most pervasive of these are the Simple Network Management Protocol (SNMP), and proprietary vendor-specific Command Line Interfaces (CLIs). SNMP and CLI have three important problems.

First, the generation of important management information that is only found in different, vendor- and technology-specific forms, such as private MIBs (management information bases, a tree-structured set of data for network management) and the many different incompatible dialects of CLIs, aggravate this problem.

Second, common management information, defined in a standard representation, is not available. While there will most likely never be a single unified information model (just as there will never be one single programming language), there must be an extensible, common modeling basis for integrating these diverse data, or else common management operations and deductions cannot be performed across *heterogeneous* devices. For this, the AMDL model [**?**] is used as a unified information model, it is described in A.

Third, neither SNMP nor CLI is able to express business rules, policies and processes, which make it impossible to use these technologies to directly change the configuration of network elements in response to new or altered business requirements [**?**]. This disconnects the main stakeholders in the system (e.g., business analysts, who determine how the business is run, from network technicians, who implement network services on behalf of the business).

The Role of Information and Data Models in Autonomic Computing

The idea of AMDL is to use a single information model to define the management information definitions and representations that *all* components that intent to communicate with the platform, will use. This is shown in Figure 2.4.

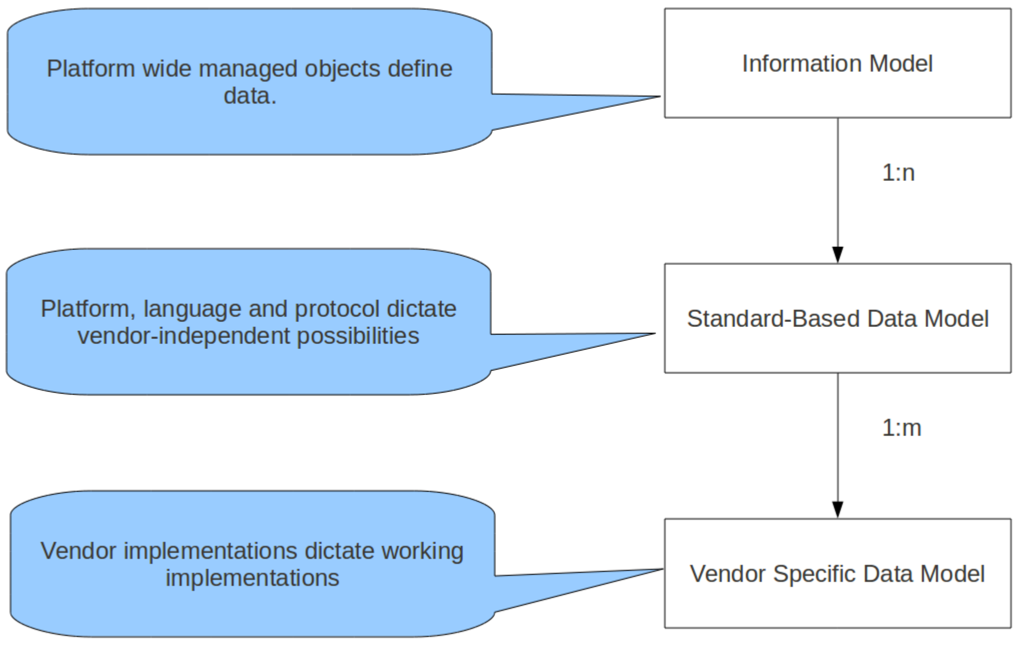
Platform wide managed objects define data.
Information Model
1:n
Platform, language and protocol dictate vendor-independent possibilities
Standard-Based Data Model
1:m
Vendor implementations dictate working implementations
Vendor Specific Data Model

Figure 2.4: Relationship between information and data models.

The top layer represents a single, platform-wide information model, from which multiple standards-based data models are derived (e.g., one for relational databases, and one for directories). Since most vendors add their own extensions, the lowest layer provides a second model mapping to build a high-performance vendor-specific implementation (e.g., translate SQL92 standard commands to a vendor-proprietary version of SQL). Note that this is especially important when an object containing multiple attributes is split up, causing some of the object's attributes to be put in one data store and the rest of the object's attributes to be put in a different data store. Without this set of hierarchical relationships, data consistency and coherency is lost.

This works with legacy applications by forming a single mediation layer. In other words, each existing vendor-specific application, with its existing data model, is mapped to the above data model, enabling applications to share and reuse data.

3. Autonomous Control and Management Platform

The previous chapter introduce the concept of autonomic computing and autonomic networking. Furthermore it pointed out the importance of an autonomic element as the building block of any autonomic system. In this chapter we present the Autonomous Control and Management Platform (ACMP)[**?**], a framework built to apply autonomic principle to control home environments.

The Autonomous Control and Management Platform (ACMP)[**?**] proposes a middleware to overcome the complexity of the different future network devices. This platform enable all the components of a network to collaborate in order to form an intelligent environment.It also provides, together with its services, the self-x functionalities, already introduced in 2.3. The ACMP proposes an unified information model called ACMP Model Definition Language (AMDL) that is explained in A, as well as autonomic knowledge distribution mechanisms.

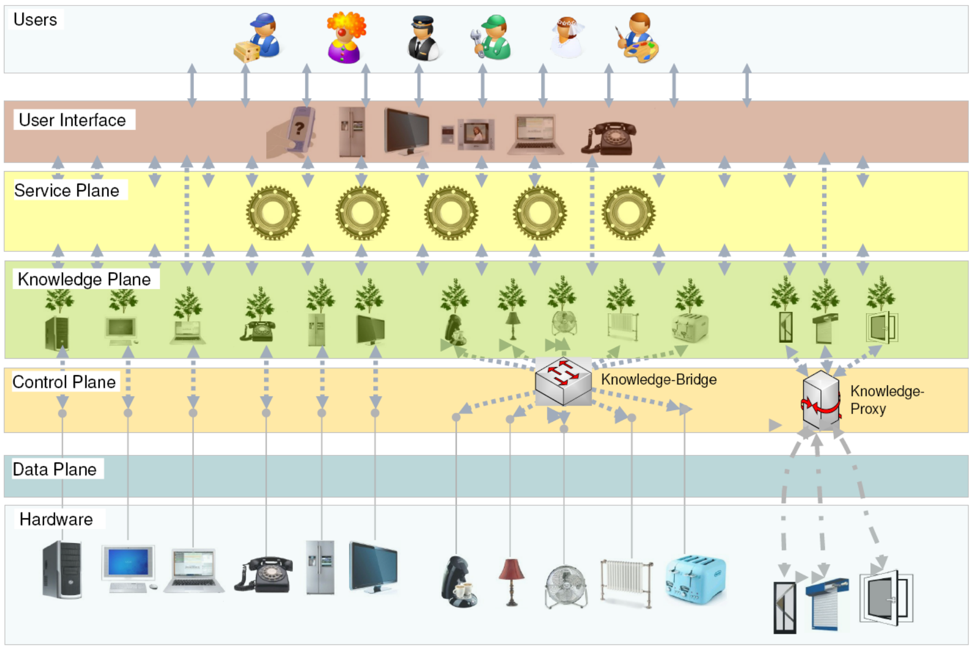

Figure 3.1: Functional planes of the autonomous control and management platform [?].

As depicted in figure 6.2, the control and management platform consists of three abstraction layers[?]:

- Control Plane (CP)
- Knowledge Plane (KP)
- Service Plan (SP)

Knowledge Plane

The Knowledge Plane(KP) is responsible for brokering information between nodes. It enables distributed use of data. This allows the provisioning of remote or distributed functionality at any place inside the network [?].

Knowledge is organized in trees. Each node connected to the ACMP exposes is represented as a *knowledge object*. The knowledge object defines the controllable aspects of a node. The interaction with the ACMP-components happens only through its *knowledge model* that is described by the knowledge object. Models resides logically inside the KP. Services inside the SP only communicate with each other and the hardware nodes through models [?].

The core component of the knowledge plane is the **knowledge agent**. It carries the major functionality and provides access to the plane for devices and services. To fulfill the task of distributing data of devices and services, those elements need to represent themselves in the knowledge plane properly. The term *knowledge* describes configuration or measuring data in combination with semantic information. For example, the information 200 `cd` is useless without some kind of semantic context information. If the value was measured outside, the implication would be that the sun is going down, so the lights on the rooms where the presence of a person is detected, should be turn on. If the value was measured inside of a room, it indicates that the room isn't probably well illuminated.

A **knowledge object** has information about the original element such as vendor information, its location, its abilities and the actual configuration and measuring data and the semantic meaning of that data. Since many devices are built from standard components, they can be represented by a *tree of knowledge objects*. Figure 3.2 gives an example for such an decomposition. The projector device is considered to be simple enough, so that it needs just one single object for representation. A laptop computer may be too complex for a single object, thus it is decomposed and uses a tree of objects for its representation. The tree's root element contains the specific data of that computer, while the children define the subcomponents of the computer and reference the root element. The children may of course be trees of knowledge objects themselves. In section 3.1 we will give a complete description of the knowledge representation.

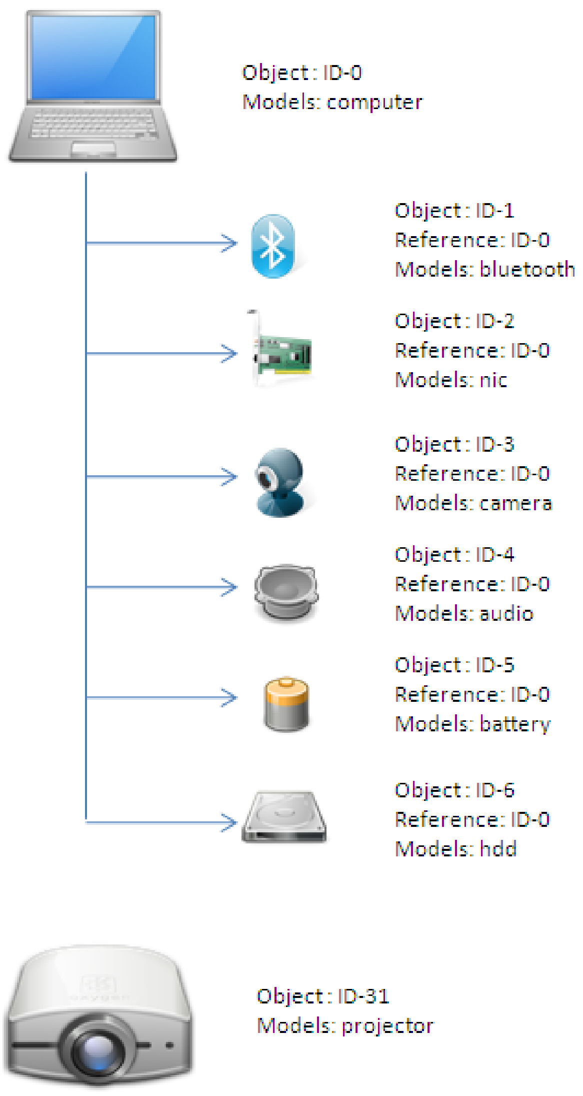

Figure 3.2: Devices representation on the knowledge plane.

The KA is also in charge of publishing knowledge events on behalf of the local publishers, and forwards knowledge events received from local or remote publishers to local subscribers.

Knowledge Agents are connected via the *Knowledge Bus* that uses the node's existing network connection as the underlying infrastructure. Services and autonomic managers on a node may contribute to the local and distributed knowledge base by producing knowledge and publishing it locally or via the knowledge bus[**?**].

The KA provides a communication interface for the local services and to the remote KAs. This interface makes possible to manage a node with a set of commands: get, set, lock, subscribe, un-subscribe. Keeping this interface simple and uniform, keeps the complexity of the networks devices manageable.

Control Plane

The *Control Plane(CP)* is in charge of the integration of the distributed node's components into the KP. Each node in the system has at least one *autonomic manager*. An *autonomic manager* is part of an *autonomic element* as seen in figure 3.3.For any autonomic application or system, the *autonomic element* is the fundamental part, which is a modular unit of composition with pre-defined interfaces and mechanisms for self-manageability.

As shown in Figure 3.3, an autonomic element typically consists of one *managed element*, which could be any computing entity that requires self-management and one autonomic manager. The autonomic element interacts with other elements and services via their knowledge agent. Each autonomic element is responsible for managing its own internal state and behavior and for managing its interactions with other autonomic elements and the environment in which it is residing.

The internal behavior of any autonomic element and its interaction with other elements is governed by the goal set by the developer of that element. The elements may be required to help other elements to achieve their goals. As it can be seen from Figure 3.3, each *autonomic manager* has four distinct functional component devoted to individual functionality of the autonomic element.

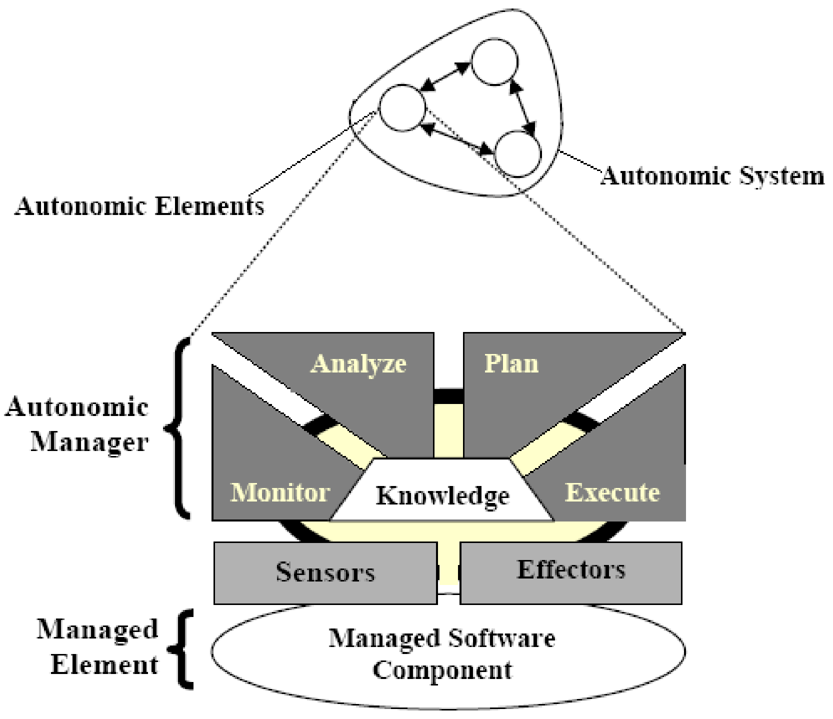

Figure 3.3: Structure of an autonomic element [?].

- The *monitor* module is in charge of extracting periodically raw sensor data. The extracted data is sent to the *analyze* module.
- The *analyze* component takes data from the *monitor* and analyzes the current operation with respect to the desired state. It is also responsible of publishing the analyzed data to the Knowledge Agent(KA).
- If any changes to be made in the behavior of the autonomic element, the *plan* component assigns task/resources based on the knowledge acquired from the KA. The *plan* is triggered by the events generated by the KA.
- Once a new plan is devised, the *execute* component performs that operation on the autonomic element or on the managed element.

The knowledge sharing process is performed either periodically of event triggered according to the needs of particular types of devices. These knowledge sharing mechanisms are meant to realize collaborative decision making processes. This is achieved by considering certain knowledge items as events that are notified to their respective subscribers. A subscriber may react to a knowledge event by performing an analysis activity and publishing the result to another knowledge event via the knowledge bus. All subscribers being notified of that result may react on that latter event and perform another analysis or a planning activity leading to an execution step [?].

Service Plane

The Service Plane is in charge of the high level management tasks. Its services are decoupled from a node's hardware by the the KP abstraction layer[?]. A service example is the control of a rooms light intensity. In order to perform that task the control service gathers the presence of any person in the room, as well as the current light intensity through the KP. This information is acquired by motion and luminosity sensors which are represented in the KP. After receiving the necessary information, the service will "decide", according to some logic, the necessary steps to fulfill the user goals, like "turn on the lights if a person is in the room". Services are able to interact with other components through the KP.

3.1 Knowledge Representation

In the previous section we defined the Knowledge Plane as one of the layers of the ACMP. This section describes the fundamentals of the knowledge objects. How to define their structure and content is presented in appendix A.

3.1.1 Concept of Knowledge Objects

A device can be represented either with a single knowledge object or a tree of knowledge objects as depicted in 3.2. The choice whether to use a single object or a tree depends on the size of the managed entity behind the service and its goals. If the managed entity has several components or is serving several goals, the tree structure yields advantages in access control and management. The service who registered the device may introduce and remove a knowledge object as it wishes. Objects consist of three types of information: identifying attributes, the object context and the data configurations.

Identifying Attributes

The purpose of the identifying attributes is to provide the information needed to exactly identify and classify an object. The mandatory identifying attributes are:

- `Object_id`: Each knowledge object has mandatory a *system-wide unique identifier* following the specification for ACMP identifiers.
- `models`: The knowledge object needs to specify the models it implements. The attribute is a list of models or prototypes. Thus it allows to search for and list objects implementing certain models. *This attribute is mandatory.*

Object Context

The `context` element of the object specifies a context valid for that specific object only. The context contains semantic information such as descriptions or the location of the service providing the object in the real world. The values within the context are an application $key \longrightarrow value$, where the key is a context-unique string and the value is a data element of any type. The actual values in the context are highly coupled with the services and the system since no exact specification of its content exists.
Examples of context values are:

Data Configurations

The `data configuration` elements contain the actual data of the knowledge object. The fundamental assumption about the nature of this data is its compatibility with a tree structure. The multiplicity of these configuration allow the services to keep several versions of the same data or publish data in partitions. This becomes useful for *autonomic managers* requiring a *DESIRED* configuration and a currently *RUNNING* configuration [?].

Key	**Value**	**Description**
location	`coordinate`	Indicates the Geographic coordinates where the device is located.
periodicity	`integer`	Defines the monitoring cycle in seconds of each field in a data configuration.

Table 3.1: Available Restrictions on Basic Data Types

4. Related Work

There have been several autonomic system projects that describe autonomic infrastructure and define the architectural aspects of an autonomic element. However these research projects address the issue of designing an autonomic element with different objectives than those presented in this thesis. The goal of this research is to make the autonomic element simple to program and use. From the perspective of implementing the fundamental autonomic element, most of these related works are instructive. However, they typically do not clarify the design of the autonomic element necessary to build an autonomic system.

4.1 FOCALE

FOCALE stands for Foundation - Observation - Comparison - Action - Learn -rEason. It was named because these six elements describe the six key principles required to support autonomic networking [**?**, **?**, **?**].

The idea behind FOCALE is summarized as follows. As discussed in chapter 1, autonomic principles manage complexity while enabling the system to adjust to the changing demands of its users and environmental conditions. In order for the network to dynamically adjust the services and resources that it provides, its components must first be appropriately configured and reconfigured. Assume that behavior can be defined using a set of finite state machines, and that the configuration of each device is determined from this information. FOCALE is a closed loop system, in which the current state of the managed element is calculated, compared to the desired state (defined in the finite state machines), and then action taken if the two states aren't equal [**?**]. This can be expanded as follows: define a closed control loop, in which the autonomic system senses changes in itself and its environment; those changes are then analyzed to ensure that business goals and objectives are still being met. If they are, keep monitoring; if they aren't, then plan changes to be made if business goals and objectives are threatened, execute those changes, and observe the result to ensure that the correct action was taken [**?**, **?**].

However, since networks are complex, highly interconnected systems, the above approach is modified in several important ways. First, FOCALE uses multiple control loops, as will be explained below. Second, FOCALE uses a combination of information models, data models and ontologies for three things: (1) develop its state machines; (2) determine the actual state of the managed element; and (3) understand the meaning of sensor data so that the correct set of actions can be taken. Third, FOCALE provides the ability to change the functions of the control loop based on context, policy and the semantics of the data as well as the current management operation being processed. Fourth, FOCALE uses reasoning mechanisms to generate hypotheses as to why the actual state of the managed element is not equal to its desired state, and develops theories and axioms about the system. Finally, FOCALE uses learning mechanisms to update its knowledge base. These differences will be explained in the following sections.

4.1.1 FOCALE's Control Loops

Figure 4.1 shows that in reality, there are two control loops in FOCALE (as opposed to one, which is pictured in most other autonomic systems). The desired state of the managed resource is predefined in the appropriate state machine(s), and is based on business goals [**?**, **?**, **?**]. The top (maintenance) control loop is used when no anomalies are found (i.e., when either the current state is equal to the actual state, or when the state of the managed element is moving towards its intended goal). The bottom (adjustment) control loop is used when one or more reconfiguration actions must be performed. Note that most alternative architectures propose a single loop; FOCALE uses multiple control loops to provide better and more flexible management. Correcting a problem may involve more actions affecting more entities than the original managed entity in which the problem was noticed.

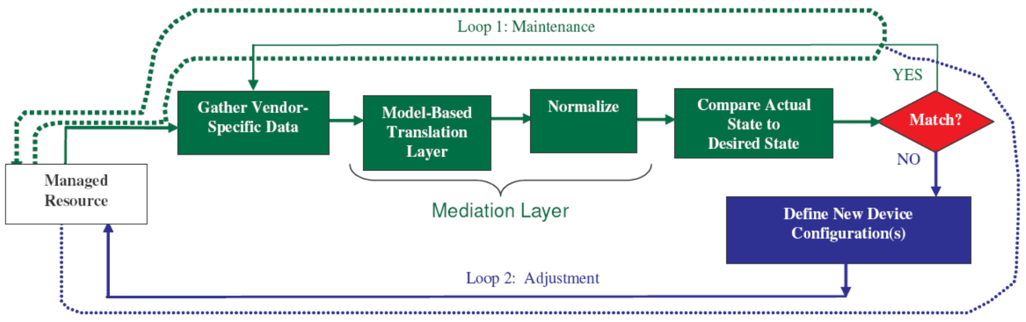
Loop 1: Maintenance
Gather Vendor-Specific Data
Model-Based Translation Layer
Normalize
Compare Actual State to Desired State
YES
Match?
NO
Managed Resource
Mediation Layer
Define New Device Configuration(s)
Loop 2: Adjustment

Figure 4.1: Simplistic autonomic control loop [**?**].

The use of two different control loops, one for maintenance operations and one for reconfiguration operations, is fundamental to overcoming the limitations of using a single static control loop having fixed functionality. Since FOCALE is designed to adapt its functionality as a function of context, the control loop controlling the reconfiguration process must be able to have its functionality adapted to suit the vendor-specific needs of the different devices being adapted[**?**].

The reconfiguration process uses dynamic code generation based on models and ontologies [**?**, **?**, **?**]. The models are used to populate the state machines that in turn specify the operation of each entity that the autonomic system is governing. The management information that the autonomic system is monitoring signals any context changes, which in turn adjusts the set of policies that are being used to govern the system, which in turn supplies new information to the state machines. The goal of the reconfiguration process is specified by the state machines, hence, new configuration commands are dynamically constructed from these state machines.

4.1.2 Integrating Models and Ontologies

An information model can be thought of as the defining document that is used to model all of the different managed objects in a managed environment. DEN-ng specifies a single information model that is used to derive multiple data models. This is required because of the diversity inherent in management information, which in turn necessitates different types of repositories to facilitate the storage, querying and editing of these data. Deriving different data models from a single common information model provides data coherency and increased manageability [**?**] .

An important point missed by most standards is that information models alone are not enough to capture the semantics and behavior of managed network entities [**?**]. Information and data models are excellent at representing facts, but do not have any inherent mechanisms to represent semantics required to reason about those facts. Furthermore, most information and data models are designed as current state models using private data models [**?**]. That is, they provide class and object definitions to model the current state of a managed entity, but do not provide mechanisms to model how that managed entity changes state over its lifecycle. IETF MIBs and ITU-T M, G are examples of 'current state' models. Autonomic and cognitive applications require monitoring the lifecycle of a managed entity, not just its current state. UML was designed to do this, but note that a significant amount of management data, such as that produced by the IETF and the ITU-T, is not in UML. The FOCALE architecture uses UML definitions augmented with ontological information to produce rich data that is then fed into machine-based learning and reasoning algorithms to avoid these problems. Hence, the FOCALE architecture is able to include legacy knowledge while providing means to edit and incorporate new knowledge.

In general, an information or data model may have multiple ontologies associated with it. This is because ontologies are used to represent relationships and semantics that cannot be represented using information languages, such as UML. For example, even the latest version of UML doesn't have the ability to represent the relationship, 'is similar to' because it doesn't define logic mechanisms to enable this comparison [**?**]. (Note that this relationship is critical for heterogeneous end-to-end management, since different devices have different languages, programming models and side effects [**?**, **?**].)

Hence, the approach used in FOCALE relies on the fusion of information models, ontologies and machine learning and reasoning algorithms to enable semantic interoperability between different models. The approach used in FOCALE is to develop a set of ontologies that provide a set of meanings and relationships for facts defined in the information model and data models. This enables the system to reason about the facts represented in the information and data models [**?**, **?**].

The autonomic manager uses the ontologies to analyze sensed data to determine the current state of the managed entities being monitored. Often, this task requires inferring knowledge from incomplete facts. For example, consider the receipt of an SNMP alarm. This is a potentially important fact, especially if the severity of the alarm is assigned as 'major' or 'critical'. However, the alarm in and of itself doesn't provide the business information that the system needs, e.g the customers that are affected by the alarm, if the Service Level Agreements (SLAs) of the customers are effected. This and other information is critical in enabling the system to decide which problems should be worked on in what order [**?**, **?**, **?**].

Given the above example, FOCALE tries to determine automatically (i.e., without human intervention) which SLAs of which customer are impacted. Once an SLA is identified, it can be linked to business information, which in turn can assign the priority of solving this problem. FOCALE uses a process known as semantic similarity matching [**?**] to establish additional semantic relationships between sensed data and known facts. This is required because, in this example, an SLA is not directly related in the model to an SNMP alarm. The inference process is thus used to establish semantic relationships between the fact that an SNMP alarm was received and other facts that can be used to determine which SLAs and which customers could be affected by that SNMP alarm [**?**, **?**].

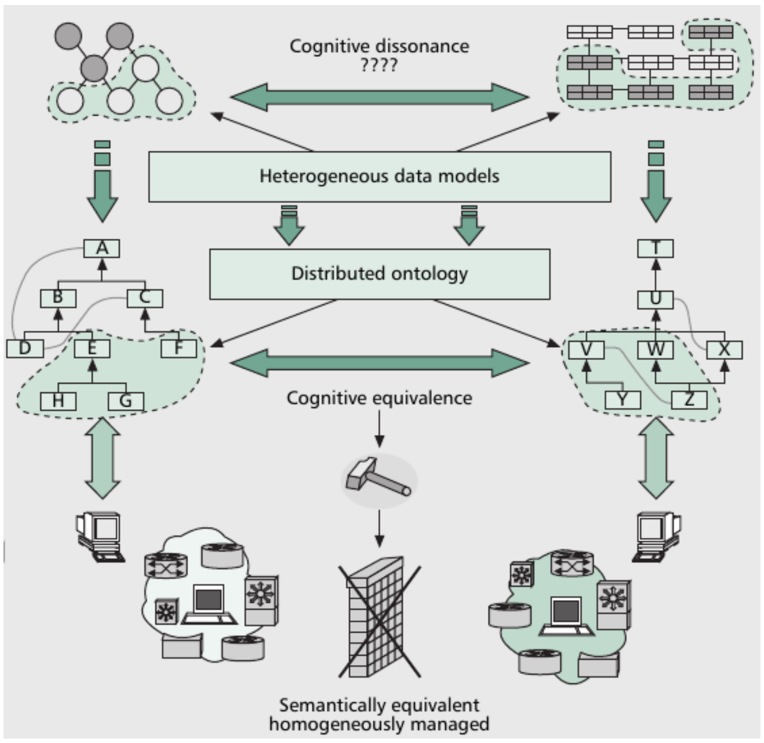

Figure 4.2: Semantic knowledge processing [**?**].

Figure 4.2 illustrates this process in use. In this figure, two different devices use two different programming languages, represented by the two different data models. They are being used to provide an 'end-to-end' service - hence, two equivalent command sets, one for each device, need to be found. Since the two devices use two different languages, there is a cognitive dissonance present. This is solved by mapping the information in each data model to a set of concepts in each ontology; then, a semantic similarity algorithm is used to define the set of commands in each ontology that best match each other.

4.1.3 Gathering Sensor Data and Sending Reconfiguration Commands

Networks almost always use heterogeneous devices and technologies. Therefore, FOCALE provides an input mediation layer to map different languages and technologies into a single language that can be used by the rest of the autonomic system. For example, it is common to use an SNMP-based system to monitor the network, and a vendor-proprietary CLI, because many devices cannot be configured using SNMP. Hence our dilemma: there is no standard way to know which SNMP commands to issue to see if the CLI command worked as intended or not. The idea of the mediation layer is to enable a legacy managed element with no inherent autonomic capabilities to communicate with autonomic elements and systems. A model-based translation layer (MBTL), working with an autonomic manager, performs this task [?, ?, ?].

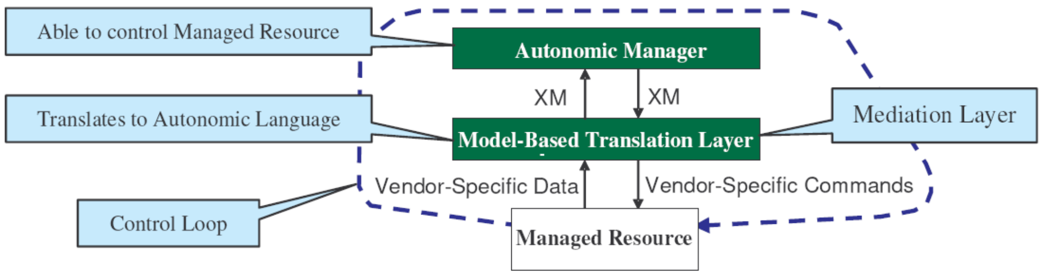

Figure 4.3: Model-Based Translation Layer of FOCALE [?].

The MBTL solves this problem by accepting vendor-specific data from the managed element and translating it into a normalized representation of that data, described in XML, for further processing by the autonomic manager. Figure 4.3 shows how the MBTL works. Two assumptions are made: (1) the managed element has no inherent autonomic capabilities; and (2) the managed element provides vendor-specific data and accepts vendor-specific commands. The MBTL enables diverse, heterogeneous managed elements to be controlled by the same autonomic manager by translating their vendor- specific languages and management information into a common form that the autonomic manager uses, and vice versa. Conceptually, the MBTL is a programmable engine, with different blades dedicated to understanding one or more languages. The MBTL translates vendor-specific data to a common XML-based language that the autonomic manager uses to analyze the data, from which the actual state of the managed element is determined. This is then compared to the desired state of the managed element (as defined by the finite state machine). If the two are different, the autonomic manager directs the issuance of any reconfiguration actions required; these actions are converted to vendor-specific commands by the MBTL to effect the reconfiguration.

4.1.4 FOCALE summary

FOCALE stands for Foundation, Observation, Comparison, Action and Learning Environment. The complete high-level FOCALE architecture is shown in Figure 4.4. This approach assumes that any managed element (which can be as simple as a device interface, or as complex as an entire system or network) can be turned into an autonomic computing element (ACE) by connecting the managed element to the same autonomic manager using the MBTL engine and appropriate blades. By embedding the same autonomic manager and the same MBTL engine in each ACE, uniform management functionality is provided throughout the autonomic system, which simplifies the distribution of its management functionality.

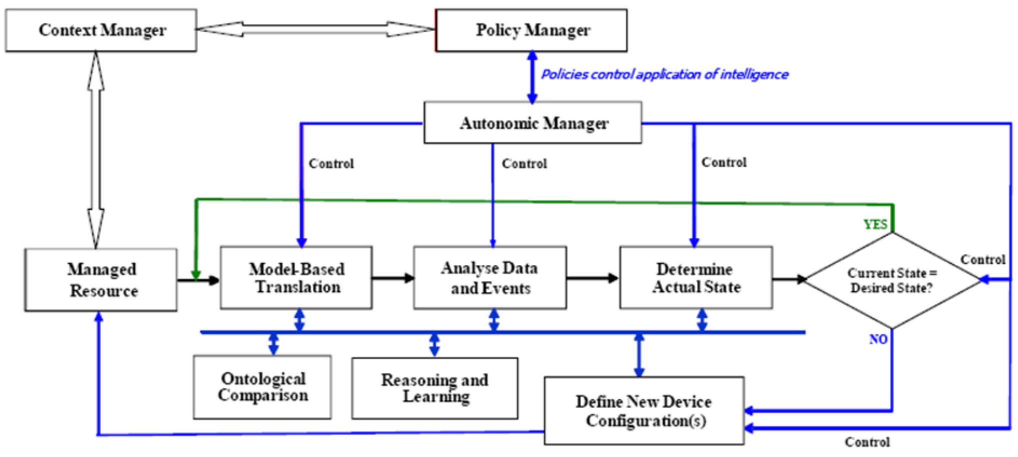

Figure 4.4: Simplified FOCALE Autonomic Architecture [?].

4.2 Ponder2

Ponder2 is a policy-based framework conceived to support autonomic computing environments. It introduces the concept of Self-Management Cell (SMC). Each cell is autonomous and facilitates the addition or removal of components, cater for failed components and error prone sensors, and automatically adapt to the user's current activity, environment, communication capability as well as interaction with other SMCs, as described in [?]. Each cell implements a local feed-back control loop, which in the ACMP is known as MAPE loop 3. An example of SMC may be the set of sensors, actuators and other devices which form a home network, although more complex devices which manage internal resources may be SMCs in their own.

A SMC is a set of heterogeneous components such as those in a home network. The interaction with each components require of different protocols. For example, interactions with PDAs, mobile phones typically occur over Bluetooth or WIFi. The SMC must have a defined view for interaction with these components for management purposes and in particular provide a uniform interface for the invocation of management actions. Therefore, adapter objects are instantiated for interaction with each component upon their discovery.

The SMC is meant as an architectural pattern that applies at different levels of scale from home networks to larger distributed and enterprise systems. Since SMCs may need to scale up to larger systems, the set of services that constitute the SMC also needs to reflect the management requirements of these systems and needs to be dynamically extensible. In [?] they assume that SMCs consist of a set of services that interact using a common publish/subscribe event bus as shown in Figure 4.5.
Although it is not necessary that all interactions be event-based, in [?] they present some advantages of this approach.

- It decouples the services since a sender does not need to know the recipients of an event, thus permitting the addition of new services to the SMC without disrupting the behaviour of existing ones. For example, a context service that gathers environment data may be added to mobile SMCs or an auditing service may be added to SMCs that require records to be kept of interactions that have occurred. Similarly, security services that perform anomaly detection, and support authentication and confidentiality as well as optimization services which try to optimize performance according to a utility function could be added in more complex SMCs.

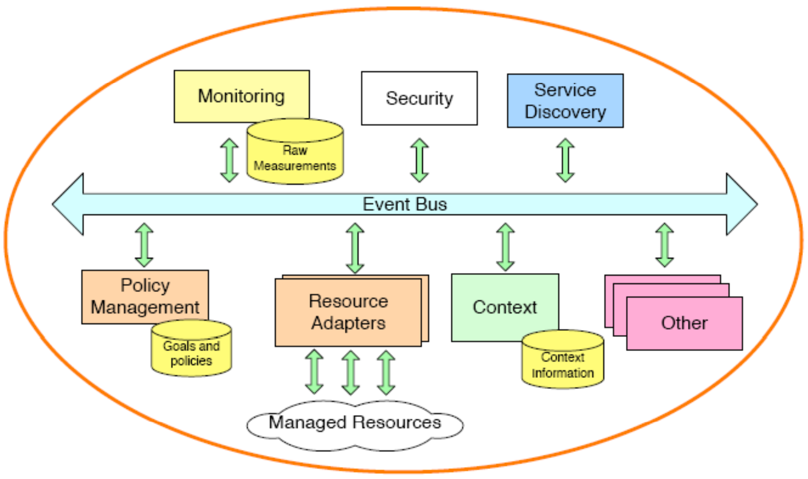

Figure 4.5: The Self-Managed Cell architecture [?].

- An event bus allows multiple services to respond concurrently and independently to the same notifications with different actions. For example, when a new sensor is discovered a policy service may initiate its configuration while a diagnostic service would take into account the additional input received from that sensor.
- The event bus can be used for both management and application data such as alarms indicating that the temperature of a room is to high.

An event may indicate discovery of a new component, component failure, change in context. It is envisioned a simple publish-subscribe event system supporting at-most-once persistent event delivery in which the service attempts to deliver the event until it knows that the subscriber is no longer a member of the SMC[?]. Interactions between management components are typically event-based in order to benefit from the extensibility they support. However, it is not specified that all interactions take place via the event bus and in particular interactions between application components can be based upon other communication paradigms such as simple point-to-point messages or remote invocations.

Figure 4.5 represents the SMC architectural pattern with an extensible set of services communicating through the event bus as well as the management and control adapters to the managed resources. Although the set of services may change depending on the context in which the SMC is instantiated (e.g., home control system, body-area network), a number of services constitute the core functionality of the SMC and must always be present. These include the *event bus*, a *discovery service* and a *policy service*. The *discovery service* is used to discover nearby components which are capable of becoming members of the SMC e.g. intelligent sensors, and other SMCs when they come into communication range. It interrogates the new devices to establish a profile describing the services they offer and then generates an event describing the addition of the new device for other SMC components to use as appropriate.

It also maintains a list of known devices as we have to cater for mobile wireless components which may come in and out of communications range and distinguish this from permanent departure from the cell.

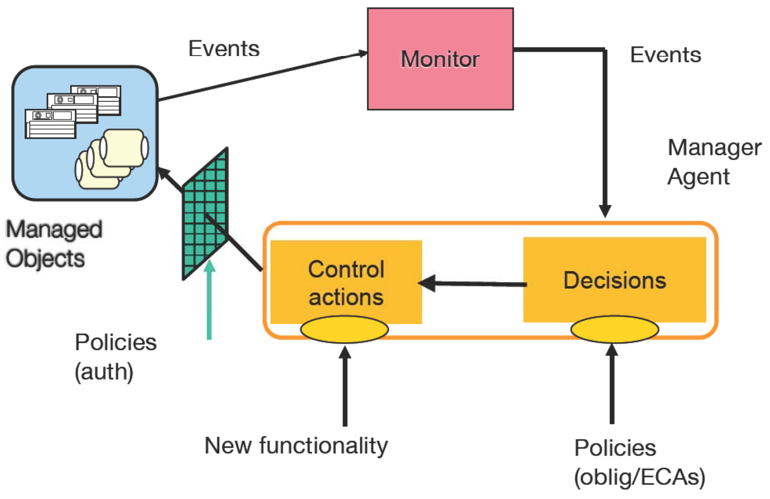

Figure 4.6: Policy-based feed-back loop [**?**].

The SMC's adaptation strategy for self-management is achieved through a *policy service* that implements a basic feed-back control loop. As shown in Figure 4.6 changes of state in managed objects are disseminated in the form of events through the *event-bus*. The policy service performs reconfiguration actions and caters for two-types of policies: obligation policies (event-condition-action rules) that define which configuration actions must be performed in response to events and authorisation policies that specify which actions are permitted on which resources or devices. Policies can be added, removed, enabled and disabled to change the behaviour of cell components without code modifications and may also be used to enable or disable other policies. For example the following policies could be specified for a home network of sensors monitoring the light intensity of a room:

```
1. on luminosity(level) do
if level < 100 then
/os.setintensity(150); /os.setMinVal(120)
2. on context(presence) do
if presence == ‘‘activated’’ then
/policies/normal.disable(); /policies/active.enable()
3. auth+ /habitant -> /os.{setfreq, setMinVal, stop, start}
4. auth+ /habitant -> /policies.{load, delete, enable, disable}
```

Policy 1 is triggered by a luminosity event as measured by a luminosity sensor and sets the intensity of the lights as well as new thresholds for the generation of events from these measurements. When the luminosity is under 100 the intensity of the lights should be 150 and an alarm should be generated if the value is below 120. Policy 2 assumes the existence of a context sensor notifying the SMC of the presence of some person in the room. When a person is present the light might be turn manually, so policies applying to the normal mode of operation should be disabled and policies specific to strenuous physical activity should be enabled. Policies 3 and 4 are the required authorizations to permit management of the intensity monitor and of the policies themselves.

The policies defined above are specified upon the instantiation of the SMC at a time when an luminosity sensor may not exist. Policies are specified in terms of roles which act as placeholders for components within the SMC. Roles are associated with interfaces, which define the methods that components must provide and events that those components can raise or that can be sent to them. This allows policies to be written in terms of the actions and events on those interfaces. When a new device, a sensor or another SMC, is discovered, it can be dynamically assigned to a role and policies defined for that role would apply to those devices. Several devices may be assigned to the same role and any policy actions would then be performed on all the devices associated with that role. In [**?**] the term mission is used for the set of policies relating to a role which is loaded onto a remote SMC that is capable of interpreting them. Sensors such as luminosity and temperature sensors monitor the condition of the room while context sensors report on the room's current activity. This is necessary in order to avoid mistaking a decreased luminosity due to a manual change on the light intensity for the sun going down.

4.3 Design based on mind agent model

Wang et al. [**?**] propose an autonomic element design based on the mind agent model. The mind agent model is defined as a six-tuple <K, A, G, P, I, L>, where:

- K is the belief knowledge base
- A is the set of behavior capabilities
- G is the set of goals
- P is the set of plan
- L is the set of policies
- I is behavior intention

Belief is the most fundamental and important element of mind agent [**?**]. Other mind states representation and reasoning rely on the rational believes. The belief can be looked as knowledge base including the basic axioms, the facts and the relevant domain data. Behavior capability describes the capabilities of mind agent and decides what actions to take, what effects will be produced after taking actions. And the capability is the basis of the mind agent plan, which decides whether the goal can be achieved. Policies including action policy, goal policy, utility policy or mixture policy, describe the rules to guide the system behaviors [**?**]. Goal reflects the state or behavior changes after executing a specified task, which is usually produced when it specifies the tasks and policies. Plan is the action plan and determines the approaches to achieve the goals. Plan connects the belief, the capability and goal together, which illustrates what actions to take for completing the specified goal based on the belief. Intention is not equal to goal. Some researchers look intentions as the choice and commitment of plan, however, in our model the result of the plan, namely the action sequence, is the behavior intention of the autonomic element. The architecture of mind agent is shown in figure 4.7.

The belief base is the bases of knowledge representation and reasoning. Action descriptions are used to express the capabilities and services of a mind agent. Each action description is

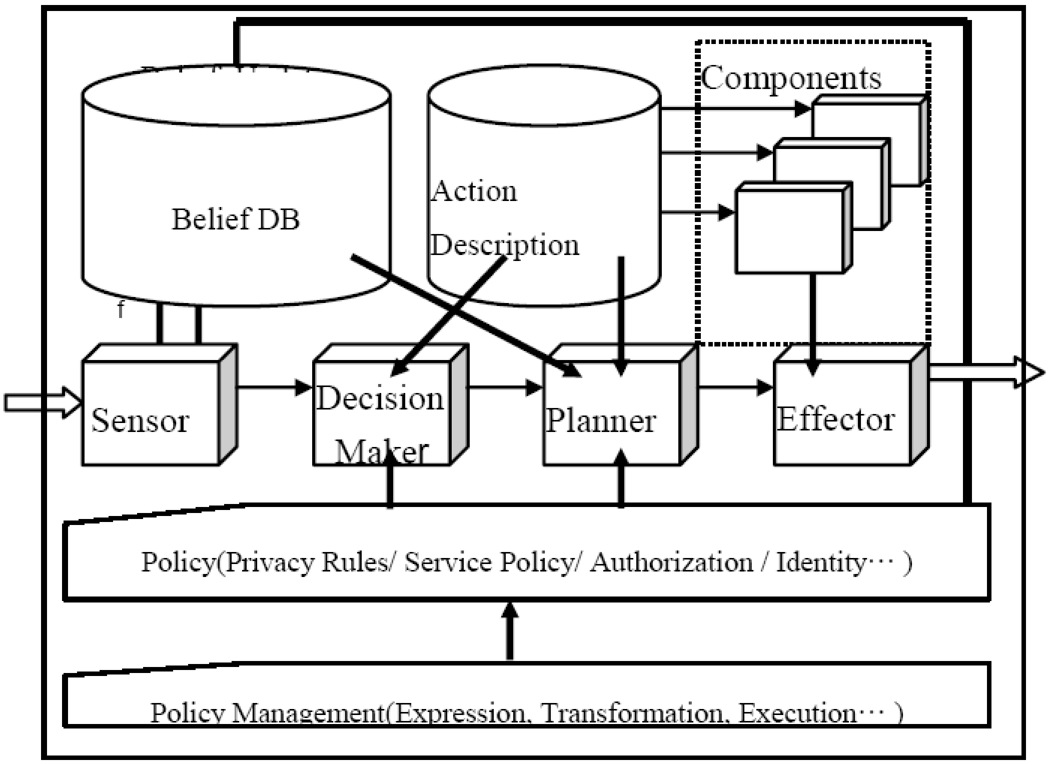
Components
Belief DB
Action
Description
f
Sensor
Decision
Maker
Planner
Effector
Policy(Privacy Rules/ Service Policy/ Authorization / Identity···)
Policy Management(Expression, Transformation, Execution···)

Figure 4.7: Mind agent model [?].

connected to a behavior entity, which is encapsulated as a component. These behavior entities act as functional components with standard interface and can be loaded dynamically [?]. Sensor can be regarded as the sense organs of mind agent. It can observe the environment and receive messages from other mind agents or users. When sensing new occurrences, the sensor will send the new observation to decision maker, also it can update the belief of mind agent[?]. Decision maker can be regarded as a reasoning machine, which will decide whether a new goal will be achieved or not. The decision will be made based on mind agent belief, observation, and policy. There are three levels of decision. The first level is directly made based on observation, which is called reflection decision. The second level of decision is based on mind agent belief and policy, which is called intelligent decision. The third level of decision is automatically made based on mental state of mind agent, such as desires, long-term goals and complex policies[?].

This is a promising and intuitive approach to meet some challenges in autonomic computing initiative; however the authors actually did not implement the design to observe its workings in a real environment.

Existing research on multi-agent systems is a rich source of good ideas about system-level architectures and software engineering practices for autonomic computing. However, these approaches lack common agreement on the exact design of the control loop. This work proposes an autonomic element design for the autonomic control and management platform(ACMP) [?]. In the next section we present the proposed autonomic element architecture.

4.4 Conclusion

This chapter presented three autonomic computing frameworks, particularly focusing on the inherent control loop of each implementation. After analyzing each of them, it is concluded that the FOCALE control loop architecture uses some concepts that will be harness in the development process of an autonomic element for ACMP.

We presented FOCALE as an architecture designed specifically to support network management. Specific attention was given to its set of novel features, including the use of multiple control loops, information and data models, ontologies and policy management.

FOCALE is a closed loop system, in which the current state of the managed element is calculated, compared to the desired state, and then action taken if the two states aren't equal. Hence, FOCALE uses separate control loops - one for when the state of the managed element is equal to (or converging towards) the desired state, and one for when it isn't. This protects the business goals and objectives of the users as well as the network operators. This approach is particularly interesting an will be used in this thesis.

5. Goals of this work

The main goal of this thesis is to provide an autonomic element architecture. The autonomic element integrates the components into the ACMP *Knowledge Plane*(see 6.2. Autonomic elements reside logically in the *control plane* described in chapter 3.

There have been several autonomic system projects that describe autonomic infrastructure and define the architectural aspects of an autonomic element, some of them already presented in chapter 4. However these research projects address the issue of designing an autonomic element with different objectives than those presented in this thesis.

The goal of this research is to make the autonomic element simple to program and use and which, can be adapted to most autonomic systems with minor modifications. From the perspective of implementing the fundamental autonomic element, most of these related works are instructive. However, they typically do not clarify the design of the autonomic element necessary to build an autonomic system.

Design goals of the presented structure of the autonomic element are as follows:

1. It should be easy to understand.
2. It should be adaptable to different component domains.
3. It should employ common software design patterns.
4. It should employ open standards in order to be free of charge.
5. It should be lightweight and should not consume system resources unnecessarily.

5.1 Scope of this thesis

The goal of this thesis is to develop an autonomic computing element to facilitate the integration of different components in an autonomic framework. This requires different issues [?] to be addressed. Addressing all of those issues and developing solutions for them in a single research project is not our purpose. Only the most important and essential issues are addressed to produce an easy and useful system within the time period of this thesis. Further analysis and experimentation of other issues will continue as future work.

Following is the scope of this research:

- Formalize the creation and management of autonomic elements. This includes, designing interfaces for user interaction with the system in particular with the ACMP framework.
- The implementation of an autonomic element to support self-management functionality.

In the next chapter we present the autonomic element architecture which is meant to accomplish the goals of this thesis.

6. Reference Architecture

The objective of this section is to propose a reference architecture for a Monitor-Analyze-Plan-Execute (MAPE) control loop. The architecture will be based on the different research projects detailed in section 4. The description will only focus on supporting the functional characteristics of the system, abstracting over aspects such as security or reliability. This has been decided in order to provide a more focused view of the architecture.

Autonomic elements are the heart of any autonomic system[?]. An autonomic element is described by two distinct parts. The autonomic manager provides the functional abilities of the element and the managed element is the entity that the autonomic manager is monitoring and controlling. All autonomic elements have a control loop (MAPE-Monitor, Analyze, Plan, Execute) that dictates the work flow of the different components of the autonomic element. Figure 6.1 shows the design for the autonomic element.

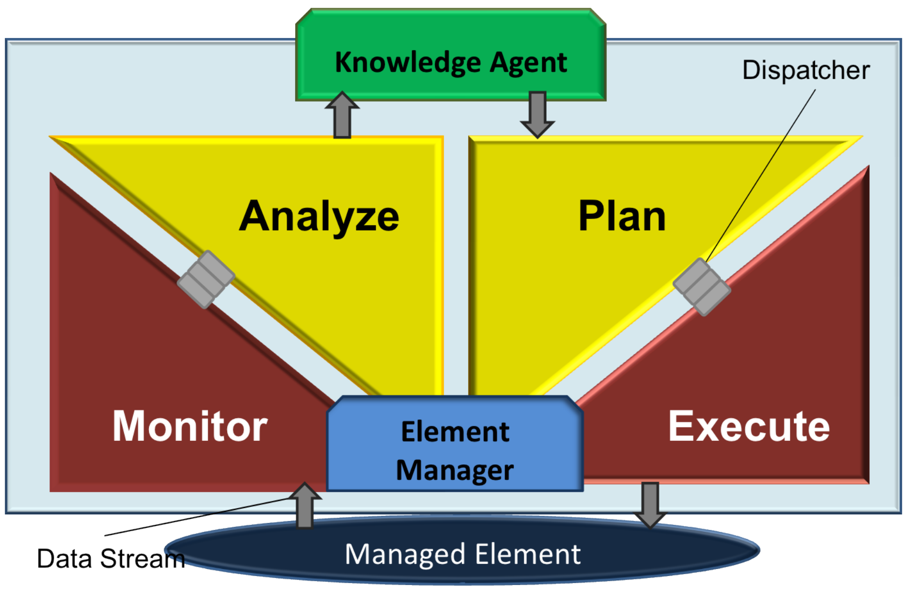

Figure 6.1: Internal Architecture of the Autonomic Element.

Before starting the architecture description, I will select a view model from the literature which enables a clear description of the architecture of the MAPE control loop. This way, I will first analyze the best known model in the literature: the 4+1 view model established by Kruchten [**?**].

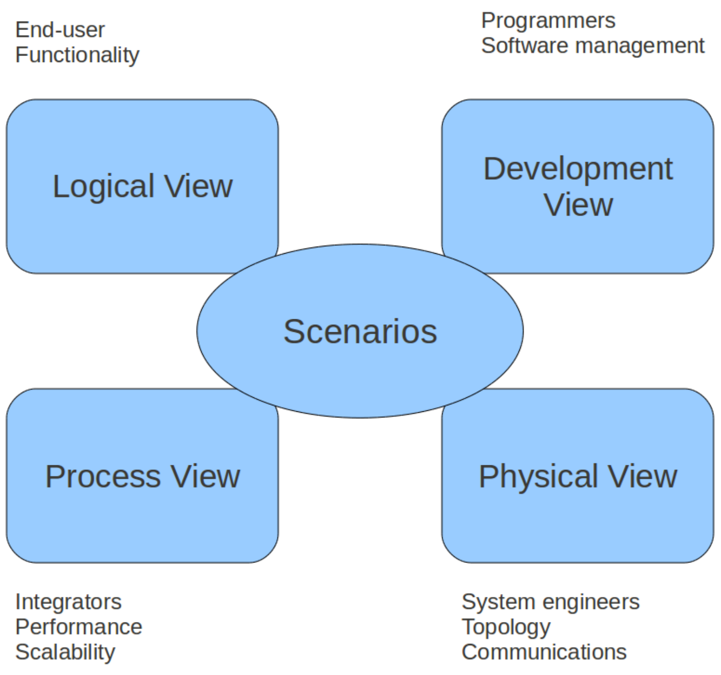

Figure 6.2: The 4+1 Architecture View Model Extracted from [**?**].

This model consists of four base views which focus on different aspects of the architecture: The logical view describes the functionality provided by the system, through the use of UML class and /or sequence diagrams. The Development view provides a logical representation of the architecture, detailing its internal structure into development packages. The Process view focuses on the runtime behaviour of the system, reflecting how the different elements interact and communicate. Finally the Physical view covers the physical deployment aspects of the proposed architecture. These four aspects of the system are complemented by the scenario view, which illustrate the architecture description through a set of use cases or scenarios, putting into context the other four views. The model allows a certain degree of flexibility, conceding that for specific cases, some of the views might not be necessary as the remaining set comprise all the important details. In this case the development view is not described, since the logical representation of the architecture is described in the process and the logical view. The 4+1 view model has the greatest acceptation among the proposals of the literature.

Once the description model has been selected, the following sections will describe each sub models, with the scenarios providing the final link.

6.1 Logical View

The logical view describes how the architecture of the system supports the required set of management functions. Whereas the details of the instrumentation subsystem are provided by the process and the development views covered in the next sections, this view focuses on the internal structure of the MAPE architecture. The description of this view will follow a top down approach, first providing a general view and describing the main characteristics of each element. The main entities of this view are the components - or functional blocks - that provide the functionality of the MAPE architecture. The model will detail those elements, describing their responsibilities, their collaboration for fulfilling the supported scenarios, and the managed information by each of them.

Figure 6.3 provides a high level view of the functional architecture. The core is composed by seven components which collaborate to provide the required operations. The Plan component contains six sub-components itself. Those entities can be seen in the central part of the picture, as well as the internal communications among them. At the top and bottom of the figure, the main elements of the domain knowledge are represented, detailing their relationships with the described components (storage, manipulation). Once the general view has been established the specific responsibilities and relationships will be detailed for each element:

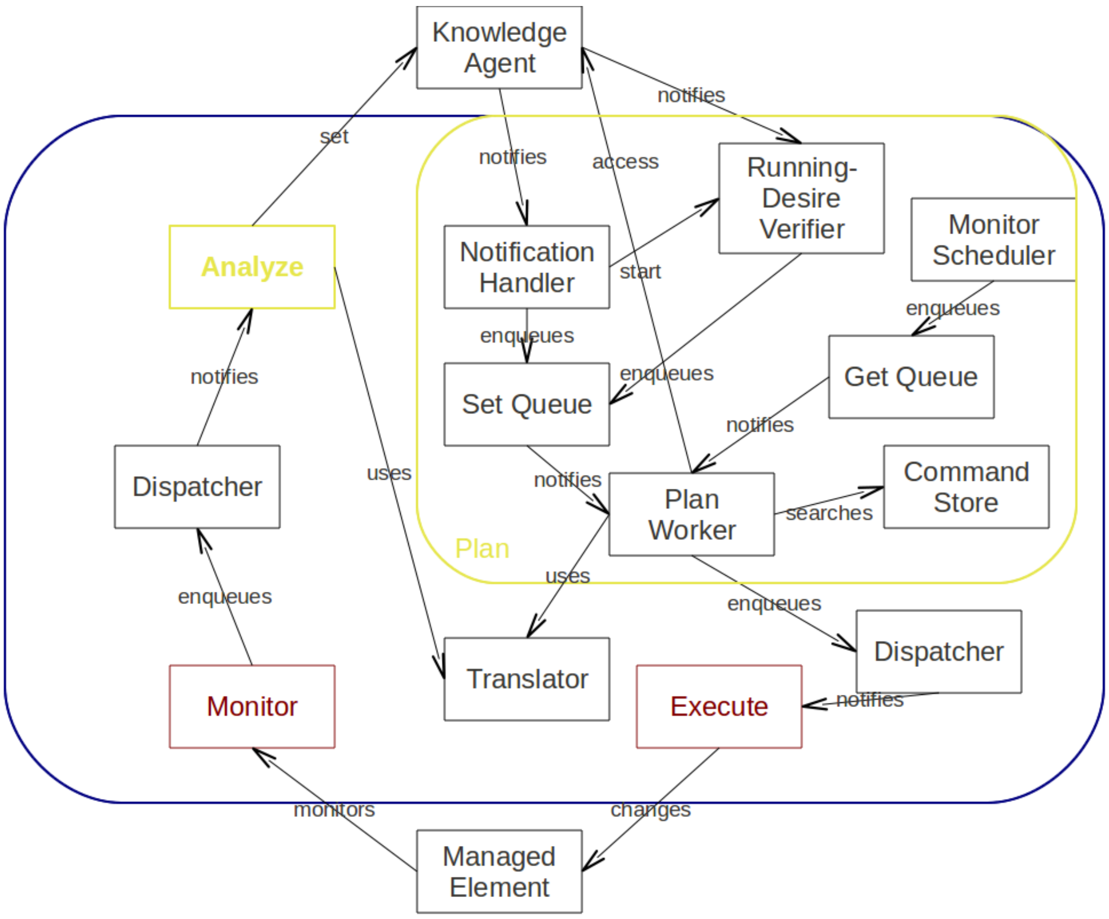

Figure 6.3: Functional Model High lever view.

In order provide a better understanding of this section, it is important to recall some concepts explained in previous chapters, like the *knowledge representation* and the concept of *data stores*. As previously described in 3.1 the knowledge plane 3 stores information about the managed elements in form of management models. A management model is a collection of *fields*. A field symbolize a characteristic of a managed element that can be monitored and configured. Each field has a state or value that in turn acts a the state representation of that particular field e.g. If the managed element is a television, one of its fields might be the channel and the value could be a number indicating the channel number. This collection of fields is ordered in a tree form. The concept of different configuration data stores was also explained in 3.1, in this section we are going to refer to the *running* and *desire* data stores. The running data store contains the actual state of the managed element and the desire data store, as its name indicates, holds the desire state of the managed element.

The knowledge agent is in charge sending notifications to the *Plan* component when changes take place in the knowledge plane. When the plan component receives a notification from the knowledge agent indicating changes, it calculates the steps needed to be followed in order to meet the changes and reflect them on the running data store. This steps are specified in form of commands that can be interpreted by the managed element. A *command* is an directive with which we can change or monitor the state of the managed element, in the next section we give a more detailed description of a command.

The internal work of the Plan module, will be explained in the next section. It is important to mention that the planer utilizes a *translator* component in order to build the appropriate commands to change the configuration of the managed element. The translator provides the same functionality as the Model Based Translator Layer part of the FOCALE architecture presented in 4.1. Its main function is to translate vendor-specific data coming out of the managed element to an AMDL representation described in 3.1. The opposite way of the translation function is also supported i.e. translating from AMDL representation to vendor-specific. After the decision about how to reach the desire state have been meet, the plan component enqueues the changes in a *dispatcher* component.

A *dispatcher* is a special data structure that resembles an operating system semaphore with added functionality. Dispatchers are placed between each pair of sub-components in the autonomic element to transfer data between them. The main responsibility of the dispatcher is to receive data from one side and notify the destination component of the data. It is important to emphasize that the design of each component in the Autonomic Element is envisioned to run in its own process, this introduces some concurrency issues which are solved by the dispatcher. Although it is desirable to keep the communication mechanism open between the individual sub-components from an architectural standpoint, dispatchers are introduced to pass information between components for two main reasons. Firstly, to avoid any deadlock situation between two concurrent components during data transfer. Secondly, to avoid wasting any processing time of the components unnecessarily. Instead of the components polling each other for data packets, the dispatcher notifies the corresponding component if there is new data awaiting processing. Polling introduces synchronization and deadlock issues and a component is expected to spend most of its allotted schedule time polling for incoming data. Having data transfer through the dispatcher improves the response time of the components and does not block the receiver if the sender is processing at a slower rate than itself for any particular reason. All push and pop operations in all dispatcher must be atomic.

The *execute* component receives one or more commands and executes them over the managed element. There are additional requirements that further complicate the internal function of execution. The change plan module only defines a set of partial orders, but provides some flexibility about the exact way commands are executed. In the case of this architecture there will be no parallel execution of activities, instead the executor will focus on preserving the stability of the managed element as much as possible. In order to do so, execution should be transactional. Considerations about a transactional executor were not taken in this architecture, but they are going to be explained in the outlook chapter.

The *monitor* retrieves the runtime information of the managed element. After a command is executed by the execute component, the output is sent to the monitor. This component filters the useful data of the execution output and enqueues it in a dispatcher connected to the analyze component.

The *analyze* entity uses the translator to convert the information provided by the monitor from a vendor-specific to an AMDL representation. After this translation takes place, the analyze sets the current state of the managed element in the knowledge plane by means of the knowledge agent i.e. it updates the desire data store with the current status. A more detailed explanation about the monitoring process is given in section 6.3.

6.1.1 Plan component

As depicted in figure 6.3 the plan component has several sub-components which are relevant to the architecture. In this section we will detail the functionality of each one of them.

First we are describing the *command store* which is the component where all available commands, with which we can change or monitor the state of a managed element, are store i.e. the command store is a collection of commands. An uml object diagram is provided in figure 6.4 as a visual description of a command model. A command has a name for identification and two important attributes the get and the set syntax. A command syntax defines how a command is constructed in order to be interpreted by the managed element, that includes the order of the fields necessary to build a command as well as any other elements. A command execution might affect more than one fields or gather information about more than one fields depending on the use, therefore a command holds a list of all fields that it can monitor named *get fields* and another list with the fields it can modify named *set fields*. A get field, different from a set field, holds a filter attribute, that describes the pattern that has to be used to filter the field value from the execution output.

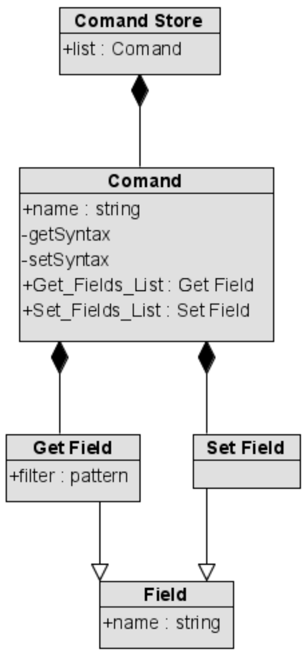

Figure 6.4: Command model.

The *Notification Handler* is the component which processes notifications sent by the Knowledge Agent indicating that something changed in the management model of the managed element. These notifications contain one or more fields whose state changed. The main function of this module consists of extracting each field which suffered a change, create a ticket that contains a field name, and enqueue them into the *Set Queue* for further processing. After a ticket has been enqueued, this component notifies the Running-Desire Verifier entity about the desired state changes. This causes the Running-Desire Verifier to start a verification process described later in this section.

A ticket is an object that has a field name, and needs to be processed by another sub-component of the plan module. Tickets are used just by the plan component. The *Set* and *Get Queues* are the components that store tickets until some other sub-component request them to do the processing job. A particularity about this components is that more than one tickets for the same field are not stored. The reason is to ensure that the state reached by the manage element is the latest one in the desire data store, and since the ticket don't hold values, there is no point in requesting more than one time a change for the same field. The set queue is responsible for storing tickets that are going to trigger some changes in the managed element configuration or behaviour, and the get queue stores all those tickets meant to retrieve the current state of the managed element.

The *Running-Desire Verifier* main function is to ensure that if a field value is changed in the desire data store, this change is reflected in the running data store. To accomplish this goal it starts a verifying process for each specific field changed in the desire data store. The process starts on waiting for the Knowledge Agent to notify about any changes of this specific field in the running data store because the moment that the running state changes, it means that the managed element has been reconfigured. When this notification is received, the component compares the value of the field from the desire vs. the running data store. If the value is the same then the process exits successfully, otherwise it issues a new ticket for this field and enqueues it into the set queue. This provokes a second try to change the state of the managed element to the desire one. The default number of attempts is set to 10, and after that the process will terminate generating an error message. In case that the process waits more than 5 seconds to receive a notification from the Knowledge Agent for this specific field, it issues a new ticket for this field and enqueues it into the get queue. With this behaviour it tries to force a notification from the Knowledge Agent, since the tickets stored in the get queue mean that the specific field will be monitored and as soon as the current value reaches the Knowledge Agent it will notify about it.

Plan worker sub-component provides an essential functionality to the plan component. It is responsible of processing the ticket contained by the set and the get queues. This component builds the appropriate command for monitor or configure the managed element, depending the ticket it is processing. In order to accomplish this task, it searches in the command store for a suitable command to perform the required task and if processing a ticket of the set queue it fetches the command needed values from the desired state of the managed element. After the values are fetched, it needs to translate the information from its AMDL representation to vendor-specific, so the command can be built and interpreted by the manged element. After the command is built, it is sent to the appropriate dispatcher in order to be further processed by the execute module. In section 6.3 we describe the processing steps undertaken by the plan worker.

6.2 Physical View

The physical view identifies the main entities of the autonomic element, as well as their base relationships. The selected view will be influenced by the main characteristics of the autonomic element, which naturally lead to identify these four main elements and their relationships.

The four main elements of the autonomic element are:

- Plan
- Execute
- Monitor
- Analyze

Each element run in a separate thread process. To deal with multi-threading environment problems, a dispatcher between each process is introduced. In the logical view, we explained the dispatcher functionality.

Figure 6.5 is an UML deployment diagram that depicts the internal physical view of an autonomic element.

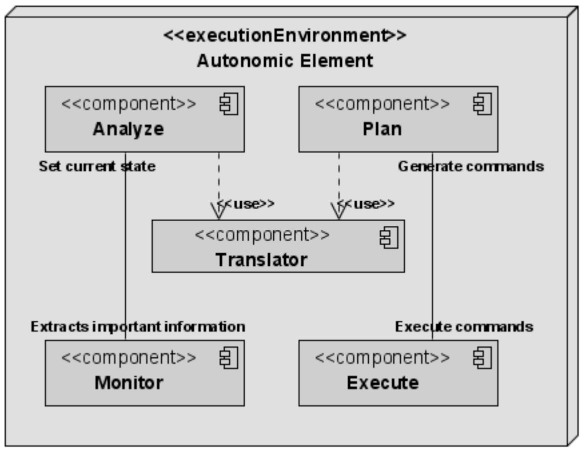

Figure 6.5: Reference Architecture Physical View.

It has already been described how the complexity and heterogeneity found in the environments is managed by the abstraction into the information models previously described. However, there is another component between the upper layer of the system, composed by the analyze and the plan module, which process those generic elements and the low-level, specific information of the managed resources. This mismatch will be addressed by a third element which acts as a bridge between them: the translator. Its role is twofold; on one hand, it must convert all the given information from the managed elements, to the AMDL model, enabling its processing by the analyzer module. On the other hand, it must be able to receive information in form of AMDL models an translate it into vendor-specific data which can be interpreted by the manged element.

In the ACMP we have distributed network nodes running a knowledge agent each. Every node provides multiple resource. Each resource has an autonomic element that integrates it with the knowledge agent. The following picture shows how the autonomic element architecture will be organized with respect to the ACMP.

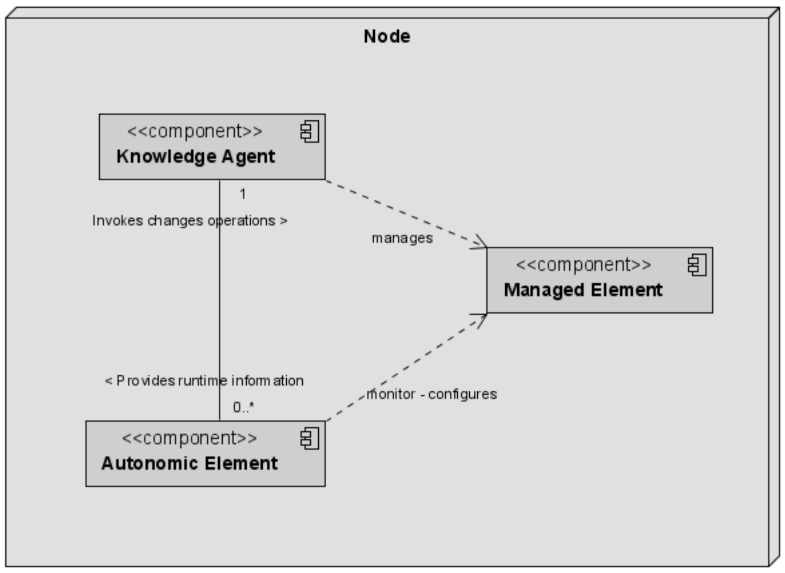

Figure 6.6: Reference Architecture Physical View.

6.3 Process View

The process view describes the information which is exchanged between the different entities, as well as the type of messages and the communication mechanisms.

The physical view has identified three main elements (autonomic element, knowledge agents, managed elements), and subsequently three potential communication channels, as shown in Figure 6.6. This way, the first step to describe the communication model will consist of analyzing the nature of these three connections. First, from the previous description it becomes clear that the knowledge agents do not directly communicate with the managed element, as every operation is executed through the autonomic element, which act as an adaptation layer between the model-based knowledge agent and the real managed elements. On the other hand, the knowledge agent-to-autonomic element communication channel plays a fundamental role, and must be thoroughly covered in this view.

In addition to the inter subsystem communications, these entities also present internal communications requiring specific mention and detail in this view. This is the case with the autonomic element which is the central focus of this thesis. Therefore, the internal organization and communication will also be covered as part of the process view description.

The following structure will be followed to describe the different facets of the communication model: First, the communication between the knowledge agents and the autonomic element will be covered, detailing how both monitoring and configuration are supported. Once these aspects have been sufficiently covered, the internal details of the autonomic element will be explained.

6.3.1 Knowledge Agent to Autonomic Element communication

The physical view has identified the two basic types of communications that occur between the knowledge agent and the autonomic element. Monitoring collects information about the managed element, providing to the knowledge agent a model of the current runtime state. Change operations consist of the knowledge agent ordering the execution of a set of changes to the managed element through the autonomic element.

Equivalently to the FOCALE architecture, we also introduce two control loops to provide the monitor and change management functions. In this section we will describe the monitoring and the change process. These two processes resemble the two control loops of the FOCALE architecture.

6.3.1.1 Monitoring Process

Monitoring operations allow the knowledge plane to keep synchronized the managed model of the managed element with the current status of the physical element. Depending on the specific scenario, there are two communication patterns for monitoring information: push and pull mechanisms. However, this architecture provides only a pull mechanism. Pull communications are initiated by the plan component, particularly by the monitor scheduler component, requesting from the managed element an updated snapshot of its information. The process consists of automatically scheduled notifications, which periodically inform the knowledge agent about the status updates of the system. This kind of notifications must be implemented with an adequate balance between the frequency of the changes (a too low frequency lowers the usefulness of the status updates) and the traffic load imposed to the network (too high frequency can cause network congestion when combined with a large environment).

In figure 6.7 we present an UML sequence diagram depicting the monitoring process. The process starts with the *monitor scheduler* which periodically creates tickets for each field present in the managed element model. As we described above, this is well known as pull mechanism. Once a ticket is enqueued in the *get queue*, it notifies a *plan worker* that a ticket is available to be processed. The plan worker retrieves the ticket and process it. The processing steps followed by the plan worker are explained in 6.3.1.2. As depicted in figure 6.7, the processing includes searching in the command store for the appropriate command and deleting all other tickets which field is affected by the selected command and and it is currently waiting in the get queue to be processed.

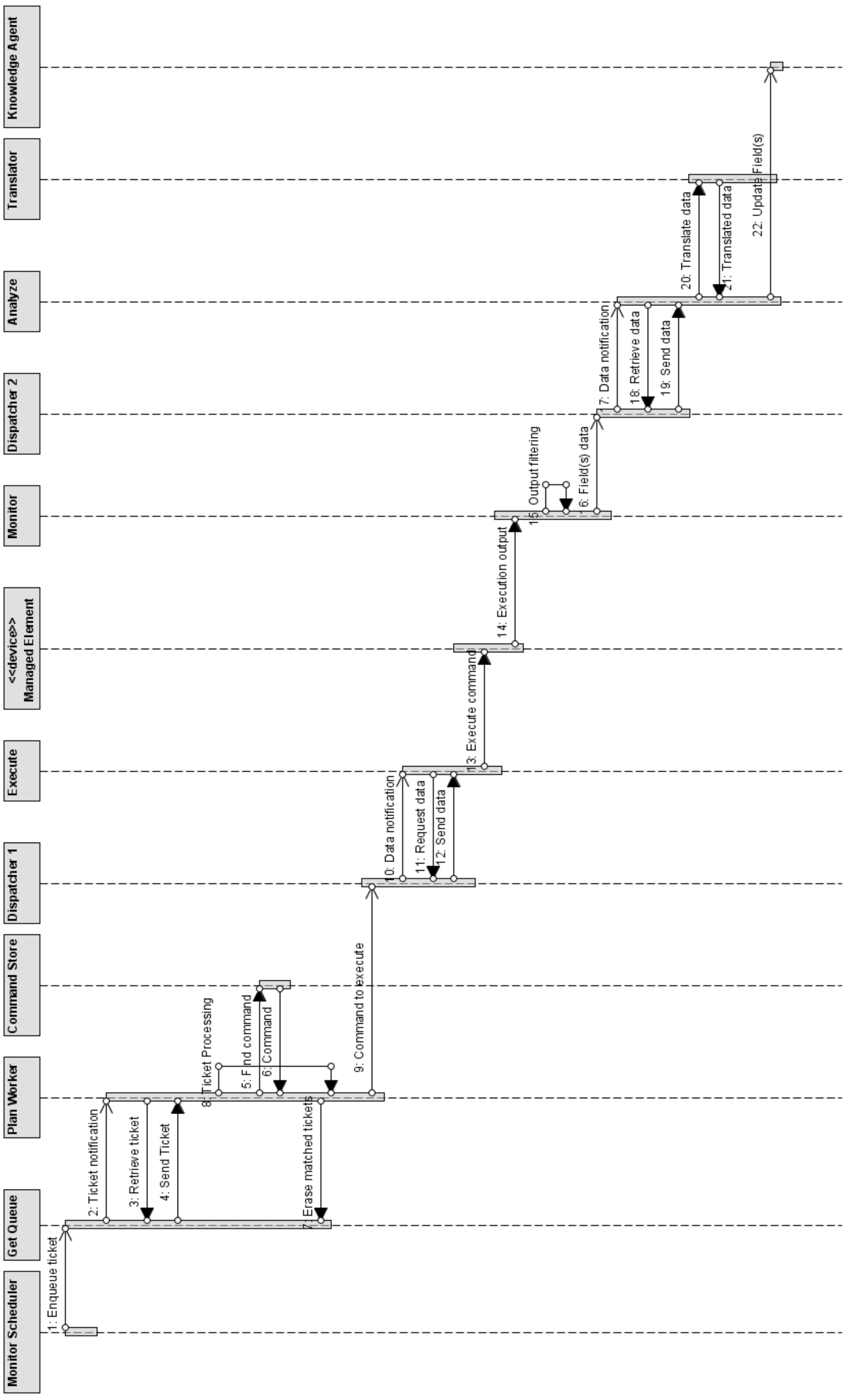

Figure 6.7: Sequence diagram of the monitor process.

When the ticket is processed, the built command is enqueued in the corresponding *dispatcher*. This last component notifies the *execute* component about new data. The execute retrieves the command from the dispatcher and proceeds to execute it in the managed element. The communication protocol of the managed element is vendor-specif, e.g. http, cli, etc. The execution output is captured by the *monitor* which filters the field(s) value using the corresponding filter pattern for each field.

The monitor enqueues the filtered data into the corresponding dispatcher. This dispatcher notifies the *analyze* component about the incoming data. The analyze retrieves the data and proceeds to translate it with help of the *translator*. Once all fields and its corresponding value have an AMDL representation, the analyze updates the current state of the managed element model by sending the information to the *knowledge agent*.

In figure 6.7 we can distinguish between synchronous and asynchronous communication forms. Asynchronous communication is depicted as a non-filled arrow. This form of communication is present between the plan-execute-monitor-analyze component communication, since each of them run in a different process.

As regards the exchange of information related to monitoring, only the messages emitted by the analyze component contain attached information, composed by instances of the managed element model, representing the current state.

6.3.1.2 Change Process

As already described in the previous section, the autonomic element architecture has two control loops, similar to FOCALE. Changing the configuration of the managed element is one of the control loops of this architecture. In this section we describe configuration change process.

Figure 6.8 depicts with an UML sequence diagram the configuration process. This process starts with the *knowledge agent* issuing a change notification indicating that the desire data store of the model of the managed element has been modified, therefore the autonomic element should configure the managed element to met such changes.

The *notification handler* module receives the change notification. It extracts from the notification the field(s) that changed. These are necessary in order to find out which command is suitable to change the manged element configuration to the desire state. After the field(s) are extracted a ticket for each field is issued and enqueued in the *set queue*. The set queue notifies the *plan worker* about the new arrived ticket(s). The plan worker retrieves one ticket at a time in order to process them. It is important to remark that the get and set queues are priority queues. When a ticket is inserted into the queue, it gets the lowest priority value. Each time a ticket is processed, all other tickets in the queue increase its priority. This is necessary to assure that a ticket does not stay a long period of time in the queue.

Plan worker processing steps are:

1. It retrieves a ticket from the queue (it can be either the set or the get queue).
2. Search in the command store for a suitable command for the specific field. In case that the plan worker is currently processing a ticket from the get queue it will search for a command that can monitor the field i.e. that the field is contained in the get fields

list. In case it is processing a set queue ticket it will look for a command capable of modifying this specific field.

3. In case that the selected command execution affects more than one field and these other affected fields have a ticket waiting in the queue where the field was extracted from, then all affected tickets for these fields will be removed of the corresponding list. This behavior leads to a faster processing of the tickets on the list.

4. Once an appropriate command is selected from the command store, the desire state of the managed element is fetched, particularly all fields value that the command needs to be built. This information is in knowledge plane, so it is accessed via the Knowledge Agent.

5. In this step the values fetched from the knowledge plane are translated to a vendor-specific representation by means of the translator component, since the command is meant to be interpreted by the managed element.

6. Last step of the process consists in sending the command to the appropriate dispatcher in order to the execute component to process it.

When the *dispatcher* receives data, it notifies the execute component about it. The *execute* retrieves this data asynchronously. The asynchronous behavior is depicted in figure 6.8 with a non-filled arrow. When the data is retrieved, the contained command is extracted and executed in the managed element. The execute component doesn't specify any fault-management mechanism, so if the configuration process fails, non corrective actions are undertaken. In the outlook section, we described the concept of the transactional executor, which should correct fault executions.

With the above described process, the autonomic element provides the capability to configure the managed element according to a desire state specified in the managed element model. In next section we described a scenario that exemplifies the work of the autonomic element in the ACMP.

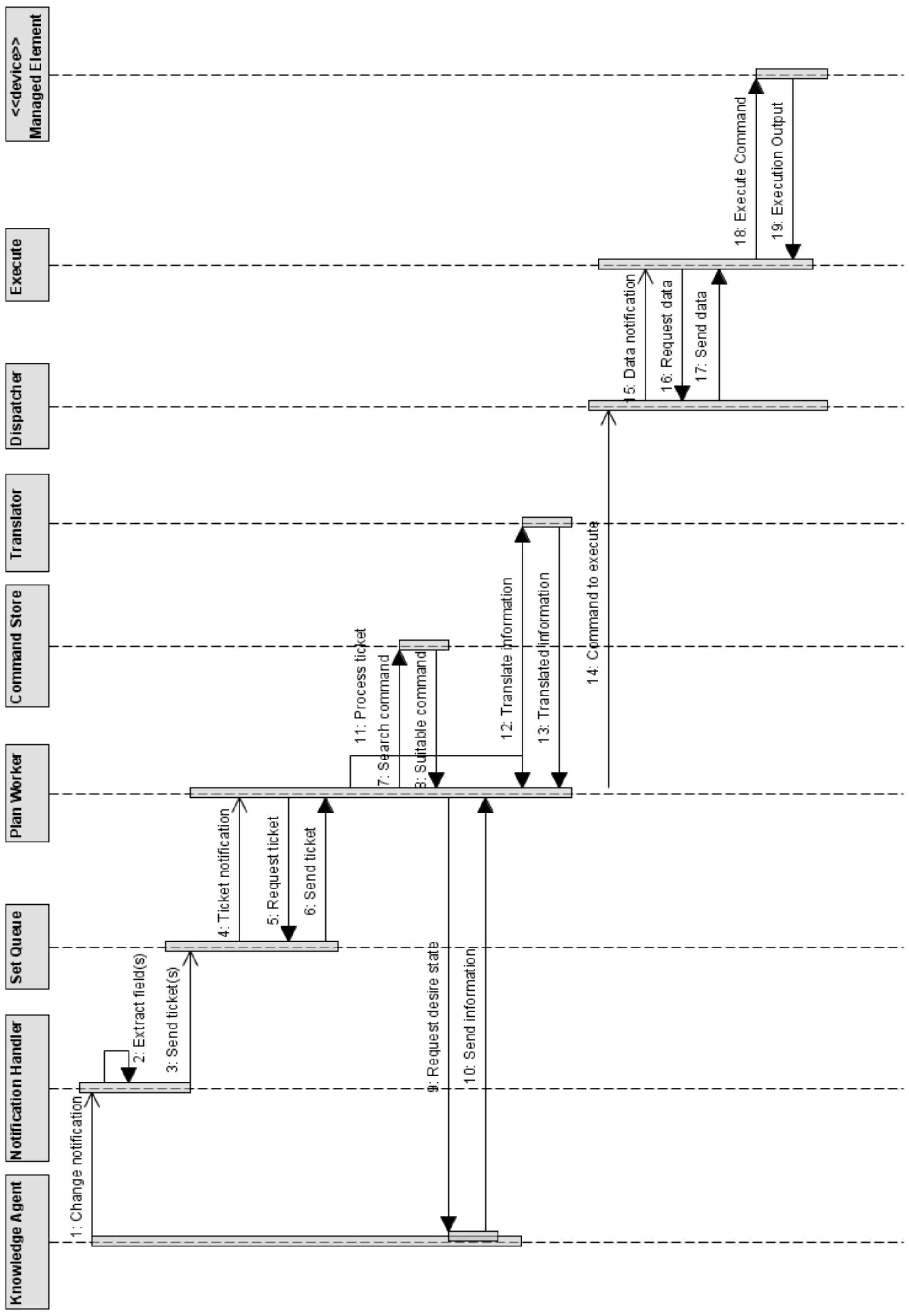

Figure 6.8: Sequence diagram of the configuration change process.

6.4 Scenario

Once the main characteristics of the autonomic element architecture have been described over the four presented views I will describe one representative scenario of the use of the autonomic element within the ACMP. It will be used as reference to clarify how the architecture operates, and the way the different previously described models allow supporting the operation of a real use case. It will first introduce the general context where it is defined, followed with the scenario description and the explanation on how the architecture supports it.

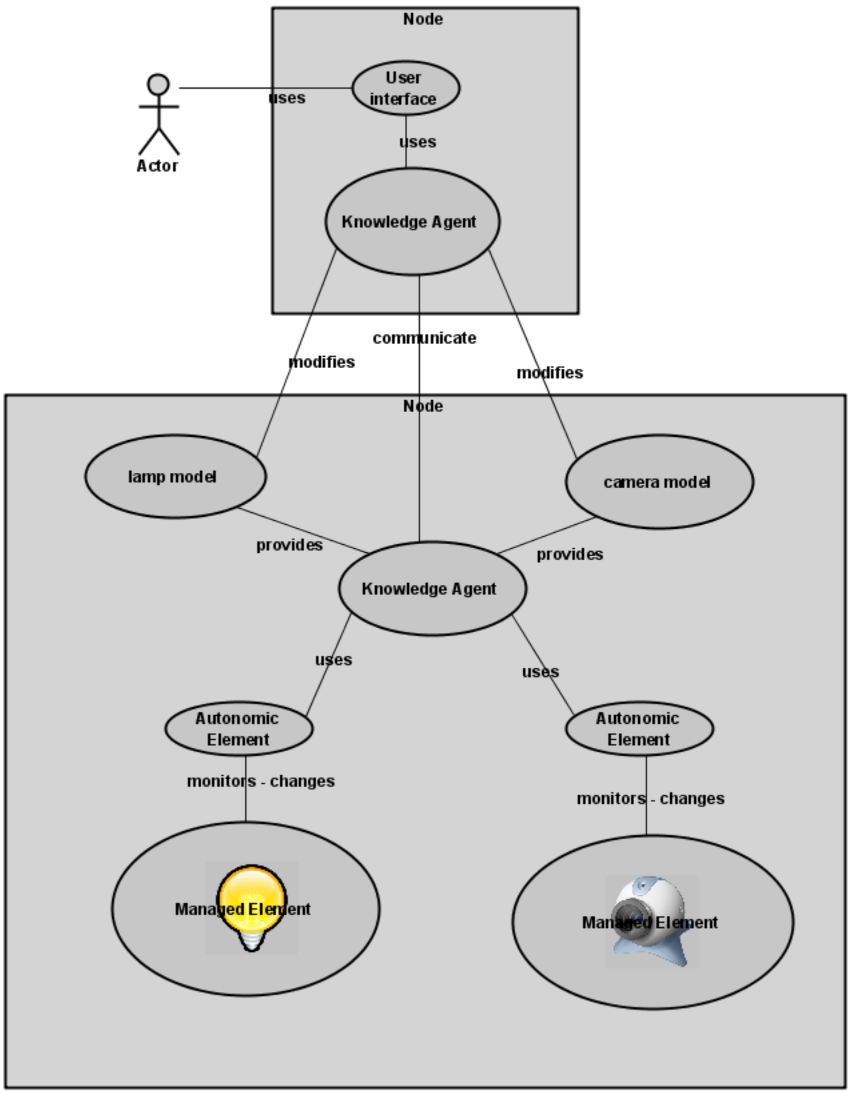

Figure 6.9: Scenario use case diagram.

In a home environment we have some devices that can be integrated to the ACMP. They are able to integrate if they provide a management interface, if they have an AMDL model, and if there exists an autonomic element that possesses the commands and translation functions needed to monitor and configure them. For this scenario purpose we selected two devices: a web-cam and a lamp. For this specific devices we assume that all requirements have been met i.e. each device provides a management interface, has an autonomic element and possesses an AMDL model instantiation.

The web-cam has the following AMDL model instantiation:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<node id="811111111111111111111111111111111111111" kox-version="v1.0" model="
    hardware/devices/multimedia/camera">
  <context>
    <location>
      <geocoordinate xmlns="http://acmp.net.in.tum.de/schema/data-types/composed-
          data-types">
        <lat>48.262967</lat>
        <lon>11.668787</lon>
      </geocoordinate>
    </location>
    <description>Example model for an ip-camera.</description>

    <periodicity>
      <state>20</state>
    </periodicity>
  </context>

  <data configuration="desired">

    <resolution>
      <columns>640</columns>
      <rows>480</rows>
    </resolution>
    <brightness>200</brightness>
    <contrast>5</contrast>
    <mode>1</mode>
    <flip>0</flip>

    <rotate>
      <p-values>up|down|left|right</p-values>
      <value>up</value>
    </rotate>

  </data>

  <data configuration="running">
    <vendor>
      <vendor-name>Links</vendor-name>
      <product-name>Pearl Camera</product-name>
      <product-number>PX-3309-675</product-number>
      <serial-number>0</serial-number>
      <revision>0</revision>
      <original-manufacturer>PEARL</original-manufacturer>
    </vendor>
    <resolution>
      <columns>640</columns>
      <rows>480</rows>
    </resolution>
    <brightness>200</brightness>
    <contrast>5</contrast>
    <mode>1</mode>
    <flip>0</flip>

  </data>
</node>
```

Listing 6.1: Camera knowledge object

The lamp has the following model instantiation:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<node id="21111111111111111111111111111111111111111" kox-version="v1.0" model="
    hardware/devices/energy:dimmable-power-socket">
  <context>
    <location>
      <geocoordinate xmlns="http://acmp.net.in.tum.de/schema/data-types/composed-
          data-types">
        <lat>48.262967</lat>
        <lon>11.668787</lon>
      </geocoordinate>
    </location>
    <description>Switch for the dimmable power socket</description>

    <periodicity>
```

```
      <state>20</state>
    </periodicity>

  </context>

  <data configuration="desired">
    <state> false </state>
  </data>

  <data configuration="running">
    <vendor>
      <vendor-name>Allnet</vendor-name>
      <product-name>All3075</product-name>
      <product-number>0</product-number>
      <serial-number>0</serial-number>
      <revision>0</revision>
      <original-manufacturer>unknown</original-manufacturer>
    </vendor>

    <state> false </state>
  </data>
</node>
```

Listing 6.2: Lamp knowledge object

In this scenario we have have two separate workstations, identified as nodes in figure 6.9. Each of them is running an knowledge agent that provides the knowledge distribution, and agent integration capabilities. As depicted in figure 6.9 one node provides a user interface, while the other one has two autonomic elements in charge of the two devices above mentioned. The translation functions and the commands for the two devices are presented in appendix B.

For the user it is transparent the knowledge and the agent distribution in the network. Using the user interface, the user is able to control the behavior of both devices, disregarding if they are local or in any other node where a knowledge agent is running.

The autonomic element of each device updates the "running" configuration periodically, this is accomplished by the monitoring process. The monitoring period of a field is defined in the model instantiation context, it is identified as the "periodicity". On the other hand, the user is able to modify the "desire" values of each field of both device model. When a field is modified, the knowledge agent notifies the affected autonomic element of the modification, and it configures the device according to the desire state. For example, if the user would like to turn off the lamp, he will set the light state value to false through the user interface. Once the autonomic element receives the notification, it starts a change process for this field. The changes "running" configuration will appear the next time that the field is monitored.

7. Evaluation

A test bed environment is developed to simulate the behavior of the proposed autonomic element in a real environment with different parameters, such as input rate and the size of a data packet. This allows observation of how components interact with each other to be made and how much time they really spend for their own management to be measured. All internal communication is performed in a standard format, named packets. Packets normally travel between different components in the direction of the control flow (Plan -> Execute -> Monitor -> Analyze). A packet is essentially a collection of information which can be interpreted by all the sub-components of the autonomic element. In executing the autonomic element, the following technique is used to optimize the operation:

- Classes for the components are designed in a modular fashion and class loading is performed only after determining that a particular module is needed. This provides a fast startup and keeps a small memory footprint when the element is not used.

All experiments were performed on an intel centrino processor (1.5 GHz) IBM Lenovo Notebook running Ubuntu 10.4. A sequential version of the autonomic element is also implemented to compare the concurrent version against. In both cases, the MAPE loop has the same amount of processing on the intermediate data packets. Since machines with multiple processors or cores are becoming increasingly common now-a-days, the concurrent version can exploit this architecture. Therefore, from our viewpoint, it is sensible to measure throughput over a fixed amount of time in both versions instead of measuring execution time of a data packet or a number of data packets. To measure throughput, both versions are ran for a preset amount of time and are both flooded with data packets at the same input rate. It is measured how many data packets both versions of the control loop can handle within that set amount of time. Figure 7.1 shows the number of data packets that both of the versions can handle for different data packet sizes. The y-axis in figure 7.1 is represented using a logarithmic scale to better signify the difference for smaller values. It is evident that the proposed concurrent version has a higher throughput than the sequential version, especially when the size of the data packets increases substantially. On average, the concurrent version processed 57% more information than the sequential version in a fixed amount of time.

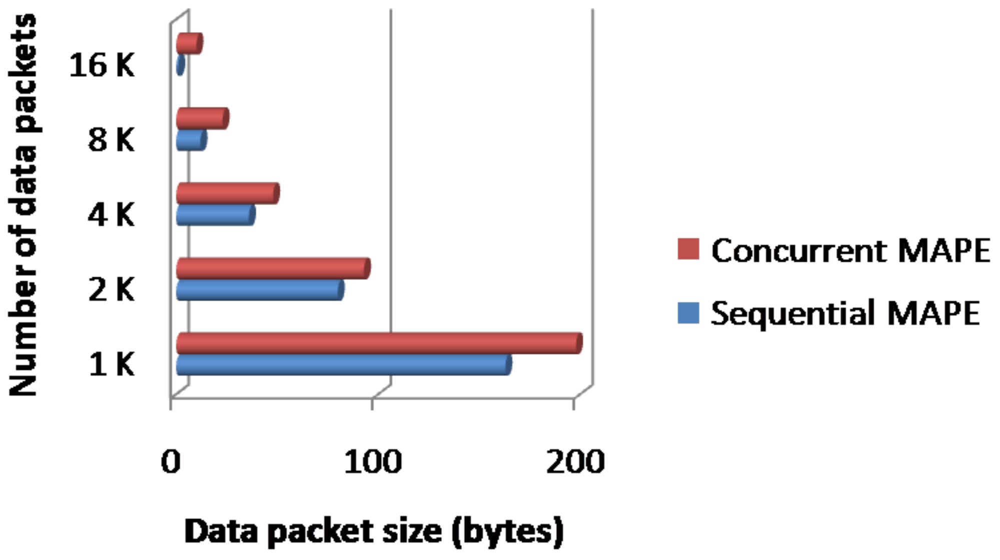

Figure 7.1: Runtime throughput.

To observe how the concurrent architecture behaves in the case of different data packet sizes and input rate, the concurrent version is executed with different data packets sizes and different input rates for a fixed period of time. Figure 7.2 shows the throughput of the concurrent model in the case of various input rates and atom sizes. From the figure, it is obvious that the model is performing as expected by handling more data packets when the data packet size is small and the number of data packets entering the system is high. The throughput decreases as the data packet size increases and the input rate decreases. With 16K data packet size, the throughput becomes independent of the input rate as our fixed period is insufficient to handle such large data packets.

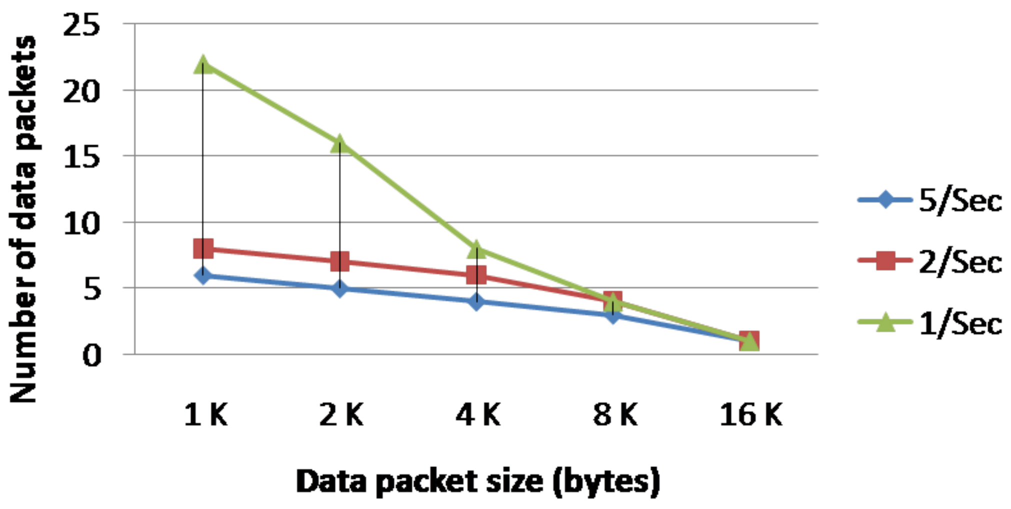

Figure 7.2: Data packet flow.

8. Conclusion and Outlook

This document goal is to present an autonomic element architecture for the ACMP platform. First we presented an introduction to autonomic computing as an initiative initiated by IBM in order to managed complexity imposed by today systems. After introducing ACMP as a control platform for house environments we focus on describing the autonomic element as the management element present in the control plane of the ACMP.

An autonomic element is the fundamental building block of any autonomic application and system. Although different aspects of autonomic computing are researched in isolation, the structural operation of an autonomic element has not been fully modeled. The standard definition for an autonomic element does not give a clear picture of how to build one from scratch and several proprietary designs have been proposed that are not inter-operable with each other. This thesis presents an engineering perspective of building an autonomic element. It is important to have a well defined model of the basic building block to develop autonomic systems. The architecture of an autonomic element has been described using the 4 +1 architecture view model, which focuses on one aspect at a time, improving the clarity of the description. The architecture has been designed with the domain-specific characteristics of the ACMP. In order to clarify how the proposal can address real management use cases, one reference scenario have been selected. The description of the scenario shows how the architecture elements collaborate to provide the require functionality to satisfy the platform functionality.

The architectural design presented is self-regulating (in the sense that multiple internal subcomponents are running in parallel but the internal data flow is consistent and unidirectional) and uses standard object oriented primitives and software engineering techniques to make it easy to develop and implement. Analysis of runtime behavior shows that it has a good design and can work smoothly without causing any deadlock or producing extensive overhead. Future works include the implementation of the same design with other programming languages. Modeling a self-healing component for the autonomic element is also considered as future work. This component capabilities includes correcting the behavior of components that aren't running appropriately of restart them if necessary.

It might be also desired to ensure transactional execution of commands in the execute component. In order to do so the execute component must automatically take back the managed element to its initial state in case there is a problem during the execution of the changes.

The executor is be able to ensure transactional execution as long as only one command is interpreted simultaneously and the following characteristics are also met. The execution of each single activity must be performed atomically, and its correct execution can be validated. If these conditions are met, after the execution of each activity the state will either be the initial one (if execution failed) or a new state with the operation successfully applied. The transactional executor will keep a stack with the successfully applied activities. In case of failure detection, the executor will automatically obtain a compensation activity for each one of the successfully executed activities, and will execute them in reverse order to cancel the changes applied by the partial execution of the plan.

A. Defining Knowledge Objects

This section shows how the structure of knowledge objects is defined. It starts with the information model and then presents the selected data model. The terms *information model* and *data model* are used as specified in [**?**].

A.1 Information Model

The information model specifies the abstract means to define the knowledge objects. The definition of the objects is separated into seven layers:

1. basic data types
2. derived data types
3. composed data types
4. models
5. prototypes
6. objects/instances
7. system

The first three layers define the data types, which cannot stand for themselves, but form the base for the models. Layer 4 and 5 then describe the base for knowledge objects. Layer 6 contains the actual knowledge objects as instances of a model or prototype. Finally layer 7 incorporates the complete knowledge in the knowledge plane.

A.1.1 Basic Data Types

The basic data types form the foundation for all definitions and are specified axiomatically. There are four basic data types, which represent disjunct groups of data types.

- **Binary**: The binary data type describes elements with the values `true` or `false`.
- **String**: The string data type specifies character sequences with a variable length.

- **Integer**: The integer data type specifies a mathematical modulo torus with discrete and exact numbers. The data model limits the modulo value.
- **Decimal**: The decimal data type specifies the set $\mathbb{R}$ with an upper and a lower limit for the numerical representation and a inherent imprecision. Numbers above the upper limit are represented by `infinity`, numbers below the lower limit are represented by `-infinity`.

The formalized representation is presented in listing A.1.

```
binary ::= (true|false)
string ::= char*
integer ::= int
decimal ::= int+.int+

basic ::= binary | string | integer | decimal
```

Listing A.1: Definition of Basic Data Types

A.1.2 Derived Data Types

Derived data types apply a restriction to a basic data type. This allows to create limited types such as `byte` or enumerations. Except for the binary type, each of the basic data type has specific applicable restrictions listed in table A.1. The definition of the `byte` type is for example:

```
byte ::= integer (lowerLimit = 0; upperLimit = 255);
```

Data Type	Restriction	Description
String		
	`length`	Limits the length of the string or defines a minimum length
	`pattern`	Defines a regular expression the string has to match
Integer		
	`lowerLimit`	Defines a lower limit for the values
	`upperLimit`	Defines an upper limit for the values
Decimal		
	`lowerLimit`	Defines a lower limit for the values
	`upperLimit`	Defines an upper limit for the values
	`places`	Defines the number of decimal places

Table A.1: Available Restrictions on Basic Data Types

These types should have a context containing semantic information about the type, but this is not mandatory.

```
ANNOTATION ::= Human-readable text containing semantic information and the author,
    version and provider of the definition
RESTRICTION ::= (key value)+

derived ::= basic RESTRICTION ANNOTATION?
```

Listing A.2: Definition of Derived Data Types

A.1.3 Composed Data Types

Composed data types combine two or more data types to a new data type. For example an IP address is represented by the concatenation of four bytes. Its definition is:

```
IPv4 ::= byte byte byte byte
```

Composed data types must provide a context with semantic information, since as observable at the example, the formal definition does not provide enough information for a conclusion what the type or its fields actually represent.

```
composed ::= (basic | derived | composed)+ ANNOTATION
```

Listing A.3: Definition of Composed Data Types

A.1.4 Models

Models are the first layer of the information model that can be used as a basis for knowledge objects. They specify the complete structure of a knowledge object and its context. The model does not contain any real data, but it has to provide semantic information about its nature and purpose. Listing A.4 shows the formal definition of models. Its additional values are `ACI` specifying the default access control, `CONFIGURATIONS` as definition of the different available data configurations (see 3.1.1) and `PARENTS` specifying the parent models[**?**].

```
datatype ::= basic | derived | composed
PARENTS ::= name(model)*

model ::= (datatype | model)+ ANNOTATION ACI CONFIGURATIONS PARENTS
```

Listing A.4: Definition of Models

A.1.5 Prototypes

Prototypes add real data to one or more models. This allows to e.g. set default values or constant data such as vendor information or technical data to the later object with a partial allocation. Its additional values are equal to those of the model [**?**].

```
VALUE ::= value of any data type
mapping ::= string VALUE

prototype ::= mapping+ ANNOTATION ACI CONFIGURATIONS PARENTS
```

Listing A.5: Definition of Prototypes

A.1.6 Knowledge Object

The knowledge object instantiates one or more models or prototypes. It contains a complete allocation of all mandatory fields in the models and prototypes.

```
knowledgeObject ::= mapping+ ANNOTATION ACI CONFIGURATIONS PARENTS
```

Listing A.6: Definition of Knowledge Objects

A.1.7 System

Finally the system embraces all knowledge objects and adds the system context. That context contains the information where the knowledge objects are located and who may access what information of the object.

```
USER ::= user identification
ADDRESS ::= address of a piece of information within the system
ACL ::= (USER ADDRESS r? w?)*
LOCATION ::= (knowledgeObject.id agent+)*

system ::= knowledgeObject* ANNOTATION ACL LOCATION
```

Listing A.7: Definition of System

A.2 Data Model

The data model defines how the knowledge plane represents the elements from the information model. This section will present the proposed modeling language and its adequacy to handle the information model.

A.2.1 AMDL specification

AMDL stands for ACMP Model Definition Language. As argued in [**?**], the available description languages do not satisfy the requirements given by the information model. The AMDL lines up to fulfill the requirements and meet the structure of the information model.

A.2.1.1 Concept of AMDL

The AMDL directly maps the information model to the data model and orients itself at object oriented programming languages. The basic data types can be seen as native data types and elements of all higher levels as *classes* with variable declarations but without methods. Prototypes add assignments of variables to the language set.

A.2.1.2 Definition Structure

All definitions share common elements and structures. Each definition has to provide a **name space** and a **name** as attributes. The name space allows to create a directory-like structure in order to simplify the organization of the model elements. The three main name spaces are `hardware` for hardware devices, `service` for software services and `management` for plane's internal objects. For example, the name space for energy devices is **hardware/devices/energy**. The name of the definition itself must be unique within its name space.

Any definition has a **preface** and a **fields** element. The preface contains semantic and management information such as the version of the definition, its provider or a description. It also contains **extends** and **import** directives[**?**].

The `extends` element contains subelements named *extend*. The values of these elements specify the full qualified name spaces and names of the data types, model and prototypes the definition inherits from. So if a definition inherits from `hardware/devices/network/NICport`, the directive is:

`<extend>hardware/devices/energy/all3075</extend>`

The `imports` element contains subelements called *import*. It also contains the full qualified name spaces and names of the type to import. To avoid naming conflicts, the import directive allows to specify a prefix for later usage:

```
<import>hardware/devices/energy/all3075</import>
<import prefix="all">hardware/devices/energy/all3075</import>
```

For the second directive the reference is `all:all3075`.

The `fields` element contains the fields of the definition. Except for the field definition in *derived data type*, all fields provide a **name** and a **type**. For the *derived data type* the field only provides a **base** attribute specifying the parent or the data type. All fields contain meta information specifying their task and purpose.

A.2.1.3 Basic Data Types

The translation for the four basic data types specified in the information model maps the types to the largest available common native types. Table A.2 shows this mapping and the descriptors in AMDL. As an example for a mapping to a programming language the table also specifies the representing Java data type. Apart from the descriptors there is no explicit definition of the basic data types.

Data Type	AMDL Descriptor	Java Type	Value Range
binary	boolean	`boolean`	`true` | `false`
integer	integer	`long`	$-2^{63} \Rightarrow +2^{63}$
decimal	decimal	`double`	$10^{-308} \Rightarrow 10^{+308}$
string	string	`java.lang.String`	UTF-8 character sequences

Table A.2: Basic Data Types in AMDL

A.2.1.4 Derived Data Types

Listing A.8 shows an example for a derived data types. The root element is called `derived-type`.

```
<extended-datatype version="1" namespace="basic" name="byte">
  <preface>
    <metadata>
      <annotation>
          This datatype defines the extended datatype byte.
      </annotation>
      <provider>TUM, 2009</provider>
    </metadata>
  </preface>

  <!--
    The byte definition contains only the anoymous field based on the
    numeric basic datatype. It is limited to 0 on the lower level(incl)
    and to 256 on the upper level (excl)
  -->
  <fields>
    <field base="integer">
      <restriction>
        <upperLimit>255</upperLimit>
        <lowerLimit>0</lowerLimit>
      </restriction>
    </field>
```

```
23   </fields>
24 </extended-datatype>
```

Listing A.8: Example for a Derived Data Type

The lines 17-22 of the listing define the only field a derived data type may contain. The base must be a basic data type. The child element specifies the restriction on the base type.

A.2.1.5 Composed Data Types

Listing A.9 shows an example for a composed data type.

```
 1 <composed-datatype name="vendor-information" version="1" namespace="hardware/
       metadata-information">
 2   <preface>
 3     <metadata>
 4       <annotation>
 5         The "vendor-information" data-type contains all
 6         information needed to
 7         tag a device or service with vendor
 8         information.
 9       </annotation>
10       <provider>TUM, 2009</provider>
11       <revision>20091215</revision>
12     </metadata>
13   </preface>
14   <fields>
15     <field name="vendor-name" type="string">
16       <metadata>
17         <annotation>
18           This field provides the name of the vendor.
19         </annotation>
20       </metadata>
21     </field>
22     <field name="product-name" type="string">
23       <metadata>
24         <annotation>
25           This field provides the name of the product.
26         </annotation>
27       </metadata>
28     </field>
29     .
30     .
31     .
32   </fields>
33 </composed-datatype>
```

Listing A.9: Example for a Composed Data Type

The lines 15-28 show the field definitions for this type. Each field has to provide a name and a type specifying its content. The type may be either a basic data type, a derived data type or another composed data type. Composed data types may inherit fields from a parent data type.

A.2.1.6 Models

Listing A.10 shows an example for a model definition.

```
1 <model name="dimmable-power-socket" version="1" namespace="hardware/devices/energy
      ">
2   <preface>
3     <imports>
4       <import>metadata-information/*</import>
5     </imports>
```

```
    <extends>
      <extend>power-socket</extend>
    </extends>
    <metadata>
      <annotation>
        The power-socket model defines a switchable and
        dimmable
        power socket with two data configurations, "desired" for
        changes and
        "running" for the current configuration.
      </annotation>
      <provider>TUM, 2009</provider>
      <revision>20091215</revision>
    </metadata>
  </preface>

  <fields configuration="desired">
    <field name="intensity" type="byte">
      <metadata>
        <annotation>
          This field allows to control the power output of
          the socket in percent.
        </annotation>
      </metadata>
    </field>
  </fields>

  <fields configuration="running">
    <field name="intensity" type="byte">
      <metadata>
        <annotation>
          This field presents the current power output in percent.
        </annotation>
        <aci>
          service-only
        </aci>
      </metadata>
    </field>
  </fields>
</model>
```

Listing A.10: Example for a Model

A model can directly be instantiated as knowledge object. This model extends the existing model `power-socket` and adds one field per data configuration. The field specifies the intensity of the power throughput. The field in the `RUNNING` configuration provides a *access control injection* specifying that only the providing service may modify this field.

A.2.1.7 Prototypes

Prototypes add default and constant values to a model. The prototype presented in listing A.11 specifies the vendor information for the ALL3076 power socket based on the model in listing A.10.

```
<prototype name="all3076" namespace="hardware/devices/energy" version="1">
  <preface>
    <extends>
      <extend>dimmable-power-socket</extend>
    </extends>
    <metadata>
      <annotation>
        This is the prototype for the Allnet ALL3076
        dimmable power socket.
      </annotation>
      <provider>TUM, 2009</provider>
      <revision>20091215</revision>
    </metadata>
  </preface>

  <data configuration="running">
    <field name="vendor/vendor-name">Allnet</field>
    <field name="vendor/product-name">All3076</field>
    <field name="vendor/product-number">62581</field>
    <field name="vendor/serial-number">0</field>
    <field name="vendor/revision">0</field>
    <field name="vendor/original-manufacturer">unknown</field>
  </data>

</prototype>
```

Listing A.11: Example for a Prototype

A.2.1.8 Objects

Listing A.12 finally presents the knowledge object. Its parent is the prototype from listing A.11, which contains the vendor information specified by the underlying model.

```

<node id="a307628c2a3a2bc9476102bb288234c415a2b01d" model="hardware/devices/energy
    /all3076">
  <context>
    <location>
      <geocoordinate>
        <lat>48.262967</lat>
        <lon>11.668787</lon>
      </geocoordinate>
    </location>
    <description>Switch for the dimmable power socket</description>
  </context>
  <data configuration="desired">
    <state>false</state>
    <intensity>200</intensity>
  </data>
  <data configuration="running">
    <vendor>
        <vendor-name>Allnet</vendor-name>
  <product-name>All3076</product-name>
  <product-number>62581</product-number>
  <serial-number>0</serial-number>
  <revision>0</revision>
  <original-manufacturer>unknown</original-manufacturer>
    </vendor>
    <state>false</state>
    <intensity>255</intensity>
  </data>
</node>
```

Listing A.12: Example for a Knowledge Object

The object context contains the coordinates of the socket, so that services can estimate where the socket actually is.

A.2.1.9 System

The system element of the information model does not have an explicit data model. The system context is distributed over all components of the knowledge plane.

B. Scenario devices

This appendix provides a proposal for the translation functions and commands to use in each scenario presented in the architecture chapter. Two devices were considered, a pearl px-3309 web-cam, and an allnet 3076 power socket where with a lamp attached. The translation functions as well as the commands are device specific.

B.0.2 Camera

```
<?xml version="1.0" encoding="UTF-8"?>
<commandStore>

  <command>
    <command_name os=""> Camera_Resolution </command_name>
    <get syntax="'http://'__device_ip__'/get_camera_params.cgi?user=admin&amp;pwd='">
      <variable regex=".+(resolution=\d+).+" index="1"> resolution </variable>
    </get>

    <set syntax="'http://'__device_ip__'/camera_control.cgi?param=0&amp;value='__resolution__'&amp;user=admin&amp;pwd='">
      <variable index="1"> resolution </variable>
    </set>
  </command>

  <command>
    <command_name os=""> Camera_Brightness </command_name>
    <get syntax="'http://'__device_ip__'/get_camera_params.cgi?user=admin&amp;pwd='">
      <variable regex=".+(brightness=\d+).+" index="1"> brightness </variable>
    </get>

    <set syntax="'http://'__device_ip__'/camera_control.cgi?param=1&amp;value='__brightness__'&amp;user=admin&amp;pwd='">
      <variable index="1"> brightness </variable>
    </set>
  </command>

  <command>
    <command_name os=""> Camera_Contrast </command_name>
    <get syntax="'http://'__device_ip__'/get_camera_params.cgi?user=admin&amp;pwd='">
      <variable regex=".+(contrast=\d).+" index="1"> contrast </variable>
    </get>

    <set syntax="'http://'__device_ip__'/camera_control.cgi?param=2&amp;value='
        __contrast__'&amp;user=admin&amp;pwd='">
      <variable index="1"> contrast </variable>
    </set>
  </command>

  <command>
    <command_name os=""> Camera_Mode</command_name>
    <get syntax="'http://'__device_ip__'/get_camera_params.cgi?user=admin&amp;pwd
        ='">
      <variable regex=".+(mode=\d|mode=Outdoor).+" index="1"> mode </variable>
    </get>

    <set syntax="'http://'__device_ip__'/camera_control.cgi?param=3&amp;value='
        __mode__'&amp;user=admin&amp;pwd='">
      <variable index="1"> mode </variable>
    </set>
  </command>

  <command>
    <command_name os=""> Camera_Flip</command_name>
    <get syntax="'http://'__device_ip__'/get_camera_params.cgi?user=admin&amp;pwd
        ='">
      <variable regex=".+(flip=\d).+" index="1"> flip  </variable>
    </get>

    <set syntax="'http://'__device_ip__'/camera_control.cgi?param=5&amp;value='
        __flip__'&amp;user=admin&amp;pwd='">
      <variable index="1"> flip </variable>
    </set>
  </command>

    <command>
    <command_name os=""> decoder_control</command_name>
    <get syntax="">
      <variable regex="" index="1"></variable>
    </get>

    <set syntax="'http://'__device_ip__'decoder_control.cgi?command='__rotate__'&
        amp;user=admin&amp;pwd='">
      <variable index="1"> rotate </variable>
    </set>
  </command>

</commandStore>
```

Listing B.1: Camera command store

```
<?xml version="1.0" encoding="utf-8"?>
<Translator>
  <variable>
    <mape>
      <name> rotate </name>
      <regex></regex>
      <amdlFormat></amdlFormat>
    </mape>
    <amdl>
      <name> rotate </name>
      <regex>&lt;rotate*&gt;(up|down|right|left)&lt;/rotate&gt;</regex>
      <mapeFormat>$1</mapeFormat>
    </amdl>
    <postProcessing>
      <value>
```

```
         <amdl>up</amdl>
         <mape>0</mape>
       </value>
       <value>
         <amdl>down</amdl>
         <mape>2</mape>
       </value>
       <value>
         <amdl>left</amdl>
         <mape>4</mape>
       </value>
       <value>
         <amdl>right</amdl>
         <mape>6</mape>
       </value>
     </postProcessing>
   </variable>

   <variable>
     <mape>
       <name> resolution </name>
       <regex> resolution=(\d+)</regex>
       <amdlFormat>&lt;resolution&gt;$1&lt;/resolution&gt;</amdlFormat>
     </mape>
     <amdl>
       <name> resolution </name>
       <regex>&lt;resolution&gt;(&lt;columns&gt;\d+&lt;/columns&gt;&lt;rows&gt;\d+&
           lt;/rows&gt;)&lt;/resolution&gt;</regex>
       <mapeFormat>$1</mapeFormat>
     </amdl>
     <postProcessing>
       <value>
         <amdl>&lt;columns&gt;320&lt;/columns&gt;&lt;rows&gt;240&lt;/rows&gt;</amdl
             >
         <mape>8</mape>
       </value>
       <value>
         <amdl>&lt;columns&gt;640&lt;/columns&gt;&lt;rows&gt;480&lt;/rows&gt;</amdl
             >
         <mape>32</mape>
       </value>
     </postProcessing>
   </variable>

   <variable>
     <mape>
       <name> brightness </name>
       <regex>  brightness=(\d+) </regex>
       <amdlFormat>&lt;brightness&gt;$1&lt;/brightness&gt;</amdlFormat>
     </mape>
     <amdl>
       <name> brightness </name>
       <regex>&lt;brightness*&gt;(\d+)&lt;/brightness*&gt;</regex>
       <mapeFormat>$1</mapeFormat>
     </amdl>
   </variable>

   <variable>
     <mape>
       <name> contrast </name>
       <regex> contrast=(\d)</regex>
       <amdlFormat>&lt;contrast&gt;$1&lt;/contrast&gt;</amdlFormat>
     </mape>
     <amdl>
```

```
      <name> contrast </name>
      <regex>&lt;contrast*&gt;(\d)&lt;/contrast*&gt;</regex>
      <mapeFormat>$1</mapeFormat>
    </amdl>
  </variable>

  <variable>
    <mape>
      <name> mode </name>
      <regex>  mode=(\d)</regex>
      <amdlFormat>&lt;mode&gt;$1&lt;/mode&gt;</amdlFormat>
    </mape>
    <amdl>
      <name> mode </name>
      <regex>&lt;mode*&gt;(\d|Outdoor)&lt;/mode*&gt;</regex>
      <mapeFormat>$1</mapeFormat>
    </amdl>
    <postProcessing>
      <value>
        <amdl>50</amdl>
        <mape>0</mape>
      </value>
      <value>
        <amdl>60</amdl>
        <mape>1</mape>
      </value>
      <value>
        <amdl>Outdoor</amdl>
        <mape>2</mape>
      </value>
    </postProcessing>
  </variable>

  <variable>
    <mape>
      <name> flip </name>
      <regex>  flip=(\d)</regex>
      <amdlFormat>&lt;flip&gt;$1&lt;/flip&gt;</amdlFormat>
    </mape>
    <amdl>
      <name> flip </name>
      <regex>&lt;flip*&gt;(default|flip|mirror|flip+mirror)&lt;/flip*&gt;</regex>
      <mapeFormat>$1</mapeFormat>
    </amdl>
    <postProcessing>
      <value>
        <amdl>default</amdl>
        <mape>0</mape>
      </value>
      <value>
        <amdl>flip</amdl>
        <mape>1</mape>
      </value>
      <value>
        <amdl>mirror</amdl>
        <mape>2</mape>
      </value>
      <value>
        <amdl>flip+mirror</amdl>
        <mape>3</mape>
      </value>
    </postProcessing>
  </variable>

```

```
</Translator>
```

Listing B.2: Camera translator

B.0.3 Lamp

```
<?xml version="1.0" encoding="UTF-8"?>
<commandStore>

  <command>
    <command_name os=""> all3076_intensity </command_name>
    <get syntax="'http://'__device_ip__'/r'">
      <variable regex="Dimmstufe␣=␣(\d+)" index="2"> intensity </variable>
    </get>
    <set syntax="'http://'__device_ip__'/r?d='__intensity">
      <variable index="2"> intensity </variable>
    </set>
  </command>

  <command>
    <command_name os=""> all3076_state </command_name>
    <get syntax="'http://'__device_ip__'/xml'">
      <variable regex="&lt;t0&gt;(\d)&lt;/t0&gt;" index="1"> state </variable>
    </get>
    <set syntax="'http://'__device_ip__'/r?b=1&amp;r=0&amp;s='__state">
      <variable index="1"> state </variable>
    </set>
  </command>

</commandStore>
```

Listing B.3: Lamp command store

```
<?xml version="1.0" encoding="utf-8"?>
<Translator>
  <variable>
    <mape>
      <name> intensity </name>
      <regex>(\d+)</regex>
      <amdlFormat>&lt;intensity&gt;$1&lt;/intensity&gt;</amdlFormat>
    </mape>
    <amdl>
      <name> intensity </name>
      <regex>&lt;intensity\s?[\w*\p{Punct}+\w*\p{Punct}+]*&gt;(\d+)&lt;/intensity&gt;</regex>
      <mapeFormat>$1</mapeFormat>
    </amdl>
  </variable>


  <variable>
    <mape>
      <name> state </name>
      <regex>(\d)</regex>
      <amdlFormat>&lt;state&gt;$1&lt;/state&gt;</amdlFormat>
    </mape>
    <amdl>
      <name> state </name>
      <regex>&lt;state\s?[\w*\p{Punct}+\w*\p{Punct}+]*&gt;(true|false)&lt;/state&gt;</regex>
      <mapeFormat>$1</mapeFormat>
    </amdl>
    <postProcessing>
```

```
      <value>
        <amdl>true</amdl>
        <mape>1</mape>
      </value>
      <value>
        <amdl>false</amdl>
        <mape>0</mape>
      </value>
    </postProcessing>
  </variable>
</Translator>
```

Listing B.4: Lamp translator