Contract No. IST 2005-034891

Hydra

Networked Embedded System middleware for
Heterogeneous physical devices in a distributed architecture

D4.11 Consolidated Embedded Architecture Report

Integrated Project
SO 2.5.3 Embedded systems

Project start date: 1st July 2006 **Duration: 48 months**

Published by the Hydra Consortium **30 September, 2010- version 1.0**
Lead Contractor: Fraunhofer FIT

Project co-funded by the European Commission
within the Sixth Framework Programme (2002 -2006)

Dissemination Level: Public

Document File: D4.11 Consolidated Embedded Architecture Report
Work Package: WP4
Task: T4.1-T4.5
Document Owner: Klaus Marius Hansen (UAAR)

Document history:

Version	Authors	Date	Changes Made
1.0	Klaus Marius Hansen	2010-09-30	Final version based on internal reviews
0.9	Klaus Marius Hansen, Mads Ingstrup, Weishan Zhang	2010-09-29	Readying for review
0.8	Klaus Marius Hansen	2010-09-28	Adaptation from ongoing journal article

Internal review history:

Reviewed by	Date	Comments
Marco Jahn (FIT)	2010-09-30	Minor changes. Approved with comments
Peeter Kool (CNET)	2010-09-30	Approved with comments

Contents

List of Figures

List of Tables

Executive Summary

This deliverable reports on the consolidated work of WP4 of the Hydra project. As such the deliverable presents both new and previous development, and aims to present a coherent view of what has been achieved. Since this deliverable is intended to give such a view and since quality scientific output has been a priority in the last year of the Hydra project, the form of the remainder of this deliverable is that of a journal article (intended for submission to IEEE Transaction of Software Engineering).

In summary, the contributions of WP4 have been in three areas:

- *Semantic services for devices*. We have realized a web service compiler, that allows for generation of stubs and skeletons for on-device (web) services. The Service Compiler operates on syntactic and semantic service descriptions. The syntactic descriptions describe the operations of a service whereas the semantic descriptions describe platform characteristics and state-based behaviour of devices

- *Semantic self-management*. Self-management is an integral part of complex Ambient Intelligence (AmI) systems. Following a three-layer reference architecture, we realize self-management by i) component control through the realization of an Architectural Query Language (AQL) and an Architectural Scripting Language (ASL), ii) change management through description of systems in a comprehensive set of ontologies, the Self-Management for Pervasive Service (SeMaPS) ontologies , and related Web Ontology Language (OWL) and Semantic Web Rule Language (SWRL) reasoning, iii) goal management through a genetic algorithms-based system state optimizer and an AI planning-based architectural change planner

- *Semantic context and resource awareness*. WP4 has developed a comprehensive set of OWL ontologies, SeMaPS, that describe context and resources of services and devices. SeMaPS is an integral part of the self-management approach of Hydra. Furthermore, runtime support for resource awareness (through Quality of Service (QoS) management) has been realized

The developed components are made available as open source.

1 Introduction

From a qualitative perspective software engineering is concerned with building software of the best possible quality given the available resources. As systems become more dynamic, following, e.g., the trend of pervasive and ubiquitous computing, a critical challenge to this goal is that a system's use or its execution environment may change. When this happens either the relative importance of a system's quality attributes or the means available to achieve them may change.

In so far as such changes are predictable, they can be addressed at design time. However here our scope is with the changes that are not. It is by nature difficult to provide representative examples of what is unpredictable, but historically we can observe examples of this: If a flaw is discovered in a security protocol the assumptions about its strength built into a system need to be changed to retain the desired level of protection; If a system is moved to a new execution environment, such as another organization with more powerful devices, its resource constraints may be more liberal while it at the same time faces stricter security requirements; an IT service provider may have customers with the same functional requirements but different qualitative requirements depending on market segment; for a data-mining solutions provider, e.g., an online brokerage may pay a premium for high performance, reliability and security, while e.g. a call centre most likely faces fewer threats and will be comfortable with slower response times.

This can be handled by adjusting at runtime the qualitative trade-offs made in a system. Doing so requires the ability to introspect and dynamically change that system's configuration, as well as to assess through metrics the value of specific quality attributes for possible configurations. In such self-management processes it is easy to imagine including new services or other resources added at runtime. This way it is possible to improve, e.g., a system's performance or security beyond the capabilities of the set of resources initially deployed with the system. In particular, self-management capabilities are becoming important for ambient and pervasive computing systems as these are becoming more widely deployed. These systems are often operated as open systems, undergoing dynamic changes where services may join or leave at any time anywhere and available system resources change dynamically.

Although important, realizing self-management is not an easy task. There are many aspects that should be considered in a self-management solution for pervasive, embedded systems, such as sensors to detect system status, actuators to effect needed changes, corresponding change management schemas and related reasoning. Furthermore, if the existing schema cannot fulfill a quality of service requirement, planning mechanisms should be used to help finding the corresponding (near) optimal configuration, and then enabling this configuration dynamically. Thus, the whole self-management process involves quite a number of different tasks which can be realized with different technologies. Correspondingly, a self-management solution must preferably support a hybrid of different technologies on an architecture level in order to integrate these technologies seamlessly.

In this deliverable, we present an architecture for self-management that enables dynamic reconfiguration of a system in order to optimize its quality attributes at runtime towards the goals set for it, even as these goals change. Our architecture is realized in the Hydra Ambient Intelligence middleware[1], an aims at covering a full spectrum of self-management including:

1. *Self-diagnosis* for devices/services, systems, and applications, including device/service status monitoring, global resource consumption monitoring, and suggestions for

[1] http://www.hydramiddleware.eu

malfunction recovery. Here we use state machines for devices and collection of runtime context (Zhang and Hansen, 2008b)

2. *Self-adaptation* including Quality of Service (QoS) based adaptation, for example switching communication protocols to achieve different level of reliability and performance, and energy awareness for adaptation
3. *Self-protection* based on the QoS requirements, e.g., choosing the most suitable security strategies for component and service communication in a global manner using security ontologies (Zhang et al., 2009b)
4. *Self-configuration* including QoS based configuration for components and energy awareness for configuration (Zhang and Hansen, 2009)
5. *Self-optimization* to realize different self-management objectives, based on genetic algorithms (Zhang and Hansen, 2009)

The remainder of this deliverable is organized as follows. First, we present the background of self-management, especially architectures for self-management (Chapter 2). We define an architectural style based on a set of objectives and constraints for self-management in our setting. We then present our concrete architecture following a three layered self-management architecture model in Chapter 3. Here, we also present the detailed design of every layer. We show how to develop a self-managed application in Chapter 4. Evaluations regarding performance and scalability of our approach are presented in Chapter 5. We discuss related work in Chapter 6. Conclusions end this deliverable in Chapter 7.

2 Architectures for Self-Management

2.1 Background and Related Work

The task of architecting self-managing systems can be approached from several conceptual perspectives, which lead to different architectures.

In one approach, inspiration is sought from nature to build systems with no explicit locus of control. An example is division of labour in a group of robots inspired by the decentralized organization of an ant-colony (Labella et al., 2006). This approach has particularly been applied to autonomic communications (Dobson et al., 2006), but systems based on it are arguably difficult to engineer (Ottino, 2004).

Another conceptual approach is to leverage traditional Artificial Intelligence (AI) by relying on explicit representation of plans as a basis for action. Although this might lead to a system with a central control unit, (e.g., an AI planner as in (Ranganathan and Campbell, 2004)) control can also be distributed such as for BDI agents (Rao and Georgeff, 1995).

A third conceptual approach is inspired by the model used to engineer control systems (Diao et al., 2005). While it is orthogonal to the two first approaches in that it assumes nothing about representation of plans, it does require a certain level of centralized control in so far as *measured* system output must be compared with the *desired* output in order to compute the control measure needed to align the two.

Kramer and Magee (Kramer and Magee, 2007) observe that many early self-managing systems followed the sense-plan-act architecture (e.g., the Rainbow architecture by Garlan et al. (Garlan et al., 2004)) in which a system, conceptually at least, continually cycles through three phases, first sensing the system state, then planning an intervention, and finally acting out the planned intervention. This was also the case for early architectures for autonomous robotics (Gat, 1998). However, the sense-plan-act architecture suffered from difficulties in maintaining precise "world models". Brook's subsumption architecture (Brooks, 1991) avoided this by following the slogan "the world is its own best model" and relying extensively on sensors, but it did not provide sufficient means to handle complexity. The three layer (3L) architecture described by Gat (Gat, 1998) combines ideas from both, in that the state-less and low complexity online control algorithms reside in the bottom layer, while the top layer employs traditional modeling and high-complexity planning algorithms, with the middle layer acting as interface between the two. This architecture has become a de-facto standard architecture in autonomous robotics.

The self-management features of Hydra follow the three layered model proposed by Kramer and Magee (Kramer and Magee, 2007) which is adapted from Gat's three-layer architecture for autonomous robots (Gat, 1998). An overview is shown in figure 2.1. The lowest layer is the component control layer. It is responsible for retrieving information about the state of the system, e.g., which/what services exist and what their states are. It is also responsible for actuating low-level change commands issued from higher layers, e.g., installing and starting a service. The middle layer is the change management layer. It is responsible for detecting situations that need to be managed, and to perform that management according to pre-determined schemes by issuing commands to the component control layer. A scheme in this context can be a plan such as a sequence of actions, or a component such as a Petri Net or rule engine which generates output events/actions as it consumes input events. The top layer is the goal management or planning layer. When a situation is detected for which there is no applicable pre-existing scheme in the change management layer, the goal management layer is responsible for computing a new scheme, or plan, e.g.,

an AI planner can be used to dynamically generate a reconfiguration plan that is sensitive to the constraints set by the current system state and policies. Ideally, high-level policies express which plans to create.

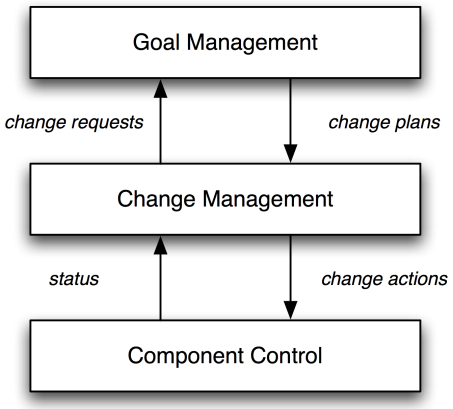

Figure 2.1: Three Layer Architecture Model for Self-Management

2.2 The Hydra 3L Architectural Style

The 3L architecture as proposed by Kramer and Magee is a logical reference model, and it does as such not constitute a clearly defined style. In the following we remedy that by defining the concrete styles that are realized by our implementation architecture.

We follow Fielding and Taylor's (Fielding and Taylor, 2002) definition of an architectural style as "a coordinated set of architectural constraints that restricts the roles and architectural elements, and the allowed relationships among those elements within any architecture that conforms to the the style.". We use the definition with the understanding that the constraints in the style are chosen to favour particular architectural objectives.

2.2.1 Objectives

The Hydra-3L style seeks to achieve the following objectives in the system as a whole:

Low coupling/Modifiability The software deployed in a pervasive computing environment is distributed across a potentially high number of devices. They have different uses and therefore diverse mobility profiles. This means their availability is unpredictable and that the software must be robust to unpredicted unavailability of devices. Further, devices frequently belong to different administrative domains, and thus the maintenance of software running on them is less coordinated and must be more loosely coupled than what is acceptable in more traditional (e.g., enterprise information) systems.

Efficiency Although the pace of innovation in hardware is high, particularly for mobile devices, older devices remain in use for long periods. This means that performance is impor-

tant, because devices are not always replaced with newer and more powerful ones. Moreover, the improvements in hardware technology are frequently a matter of making devices smaller rather than more capable, which does not change their properties as deployment platforms.

Scalability As innovation in hardware makes devices cheaper, smaller and therefore more numerous, it is important that the middleware can continue to function at larger scales.

Extensible behaviour To increase the range of its potential use, middleware must make as few assumptions as possible about the application domain while still providing useful support to application developers. For self-management this implies that the means made available for developers to specify self-management schemes for their applications must be expressive and powerful.

2.2.2 Constraints

When defining our architectural style, it is important to be clear about whether the individual design objectives, constraints and assumptions apply to the middleware itself, or to the set of applications that can be built with it. Figure 2.2 shows the conceptual relationship between the two at runtime. We use the term *constraint* when they apply to our middleware, and

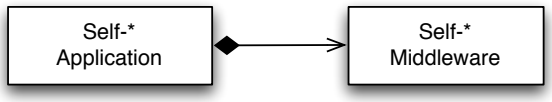

Figure 2.2: The conceptual relationship between the middleware enabling self-management, and the systems built with it.

the term *assumption* when the constraint must be satisfied for the system as a whole. The following constraints and assumptions are set by the Hydra-3L style:

- LS Layered. (constraint).
- PS publish/subscribe (constraint)
- SOA: service oriented (assumption)
- IS - semantic introspection- knowledge about the system is explicitly categorized and modelled in ontologies to enable automated reasoning (constraint)
- SAR- sensor and actuator based reflection in the component control layer. (constraint)

The relationships between the constraints and the objectives are shown in Figure 2.3 and explained below.

Layered Organizing the system into layers helps reduce coupling among components since a layer can only communicate with layers below and immediately above itself. It also helps managing complexity by introducing a higher level of abstraction, the layer, which encapsulates well-defined behaviours responsibilities. The three layers are those already described in section 2.2.

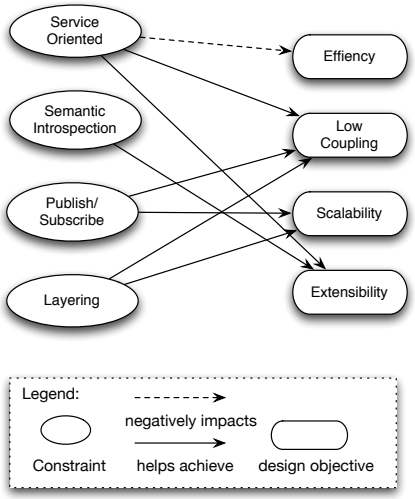

Figure 2.3: The relationships between the constraints in the architectural style and the design objectives they affect.

Layering may also help achieve efficiency because fast algorithms can possibly be executed independently of slow ones. So real-time performance, implemented in the component control layer, is not obstructed by high-complexity algorithms for optimization in the planning layer.

Publish/Subscribe Using the publish/subscribe communication paradigm for inter-component communication promotes low coupling among components by decoupling them with respect to time, space (distribution), and synchronization (Eugster et al., 2003).

Semantic Introspection Self-management algorithms operate on knowledge about the system under management. In order to manage this in a principled way and facilitate inductive reasoning, introspective information is categorized according to well defined ontologies.

Service-Oriented Architecture Both the system under management and the middleware follows Service-Oriented Architecture principles. SOA (Erl, 2005) is appealing as it promotes reuse, modularity, composability, componentization and dynamic interoperability in a standardized way. It enables the architectural principles of service encapsulation, loose coupling, service autonomy, and service discovery through well-defined service contracts and service abstraction. This both helps promote modifiability. Further, a number of existing techniques for self-management assumes a service-oriented architecture.

Sensor- and Actuator-Based Reflection To avoid the complexity and overhead of maintaining a complete and centralized system model, the introspection implementation should be based on distributed sensors which can be queried to retrieve only the metadata that is actually used, and only at times when it is actually required. For the intercession aspect of reflection (that allow modifications), the actuation of a change must necessarily be based on actuators present on the local devices where changes should take effect. Requiring these

to have a uniform interface makes it simpler to realize principled coordination of system-wide changes. Supporting runtime changes directly helps to achieve modifiability. It does so directly as it provides a way to modify a system, but more importantly because it means configuration details can be externalized rather than hard coded into the individual components. Thus the same components can more easily be used in a variety of configurations.

3 Design of Self-Management in Hydra

We describe the architecture of the Hydra self-management architecture using the approach of Clements et al. (Clements et al., 2003). Our overall system model is shown as a deployment view in Figure 3.1. The deployment follows Kramer's and Magee's model with each layer deployed on a node, and uses a publish/subscribe implementation ("Event Manager") as an implementation of an event connector and it uses an OWL ontology ("SeMaPS") as a data connector. We now describe each of the three layers in more detail.

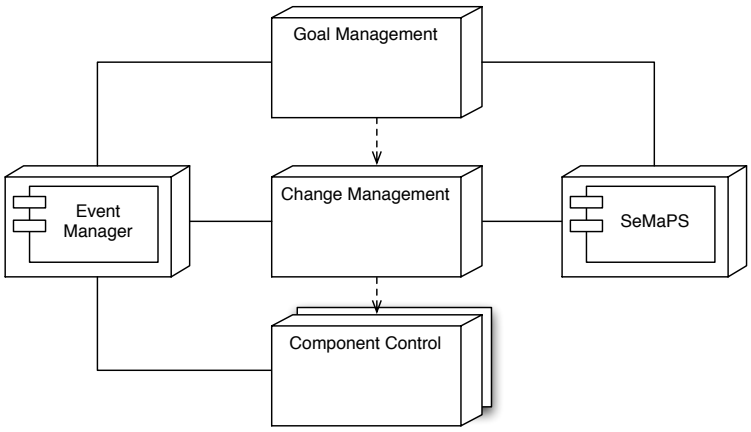

Figure 3.1: Hydra Self-Management System Model

3.1 Component Control

The Component Control layer contains elements that provide the functionality of a system. As such, in Hydra, it is composed of communicating, domain- and application-specific web services (Hansen et al., 2008b,c) that internally are structured as components. Typically, these components are implemented using OSGi (Kriens, P. (Ed), 2005). Furthermore, the layer contains facilities to interact with upper layers. This interaction is made in two ways:

1. Through *sensors*, sensing the current state of the functional elements, e.g., reporting if the current configuration of elements cannot meet design goals
2. Through *actuators*, allowing upper layers to change the current configuration so as to meet design goals

Sensors in Hydra may be one of three types:

1. *State sensors* monitor the internal state of a service. Typically embedded services fulfill a state machine specification, and we allow state changes to be reported to upper layers via events
2. *Communication sensors* monitor the communication between services, i.e., which messages are sent between which services

3. *Configuration sensors* monitor the configuration of services and components implementing these services in a system

The state and communication sensors are generated by the Hydra Service Compiler (Hansen et al., 2008b), the state sensing based on a description of a UML state machine for the service using OWL. We extend Web Service Description Language (WSDL) descriptions of web services to reference an OWL description of the service including a state machine description and use this when generating web services.

Figure 3.2 shows an excerpt of a WSDL description of a thermometer (TH03). The thermometer service defines an operation `getTemperature`, but furthermore defines a reference to a device description (in `<hydra:binding/>`). In Figure 3.3, an excerpt from the referenced OWL description of the device is shown. The device ontology also forms a basis of the SeMaPS ontologies (see Section 3.2).

```
1  <binding name="TH03SOAP" type="ns:TH03Port">
2              <hydra:binding device="file:./resources/Device.owl#PicoTh03_Indoor"/>
3              <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
4              <operation name="getTemperature">
```

Figure 3.2: WSDL description of thermometer service

The OWL description defines the TH03 device as a Thermometer (which is in itself a Sensor), references a general device description (in `<info/>`), a description of the hardware (in `<hasHardware/>`), and a description of a state-machine for the device service (in `<hasStateMachine/>`).

```
1  <Thermometer rdf:ID="PicoTh03_Indoor">
2    <deviceId rdf:datatype="http://www.w3.org/2001/XMLSchema#string">PicoTh03_Indoor</deviceId>
3      <info rdf:resource="file:./resources/Device.owl#PicoTh03_info"/>
4      <hasHardware rdf:resource="file:./resources/Hardware.owl#PicoTh03_hardware"/>
5      <hasStateMachine rdf:resource="file:./resources/StateMachine.owl#PIcoTh03_Indoor_sm"/>
6  </Thermometer>
```

Figure 3.3: OWL description of thermometer device

The state-machine for a device service plays a central role in self-management in that we assume that managed services have a description of (relevant) state changes of the device service. Our Service Compiler is able to generate support code that enables devices to publish relevant state changes as described in their state machine. Figure 3.4 shows a simple state-machine for the thermometer device service: After starting, the thermometer may go into a measuring loop after which it may stop. For example for self-diagnosis purposes, it may be relevant to know which of these states a thermometer is in[1].

The state-machine is furthermore described in OWL. An excerpt of this state-machine is shown in Figure 3.5. The description format is essentially that of Dolog (2004) and the excerpt shows part of the description of the measuring state.

Communication sensing is done through an Architectural Query Language (AQL; (Ingstrup and Hansen, 2005)) component that allows for upper layers to query the architectural structure (see Section 3.3).

Finally, actuating is performed by a component implementing the Architectural Scripting Language (ASL; (Ingstrup and Hansen, 2009; Hansen and Ingstrup, 2010)). An ASL script

[1] State-based diagnosis is not available in the open source release

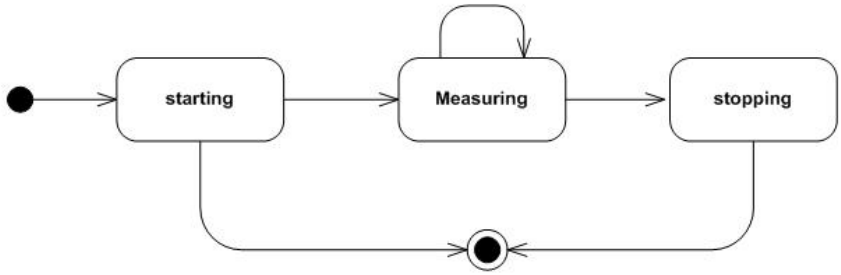

Figure 3.4: State machine for thermometer device

```
1  <Simple rdf:ID="PicoTH03_Indoor_Measuring">
2    <doActivity>
3      <Action rdf:ID="getTemperatureIndoor">
4        <Label ...>getTemperature</Label>
5      </Action>
6    </doActivity>
7    <Label ...>Measuring</Label>
8  </Simple>
```

Figure 3.5: OWL description of state machine for thermometer device

is a sequence of architectural operations, each operating on architectural elements such as components (deploying, starting, stopping, updating, undeploying). We have implemented a binding to OSGi that allows for architectural scripts that change an OSGi deployment. An example ASL script is shown in Figure 3.6. The script initializes the variable `aql_s` to reference a service from the AQL component (line 1-3), stops the service (line 4), and eventually uninstalls the component (line 5). ASL plays an integral role in reconfiguration as effected by the upper layers in our architecture. We will return to this in Section 3.3.

```
1  init_device(local);
2  init_component(aql, &SymbolicName=selfstarmanager_aql);
3  init_service(local, aql, aql_s);
4  stop_service(aql_s);
5  undeploy_component(aql);
```

Figure 3.6: An ASL Script

In summary, Figure 3.7 shows a deployment view of the Component Control layer of Hydra self-management. Furthermore, Figure 3.8 summarizes a component & connector view of the interaction among components with focus on the component control layer

3.2 Change Management

The Change Management layer is concerned with reacting to changes in state in the Component Control layer, changing parameters and behaviours according to a set of plans. In our case, plans are realized using OWL ontologies with behaviours represented as SWRL rules. The set of ontologies used in Hydra is called *SeMaPS*. The structure of the SeMaPS ontologies is shown in Figure 3.9. Our SeMaPS runtime implementation uses the Protégé

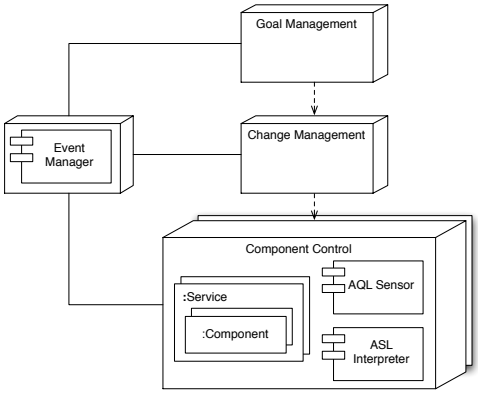

Figure 3.7: Deployment View of Component Control Layer

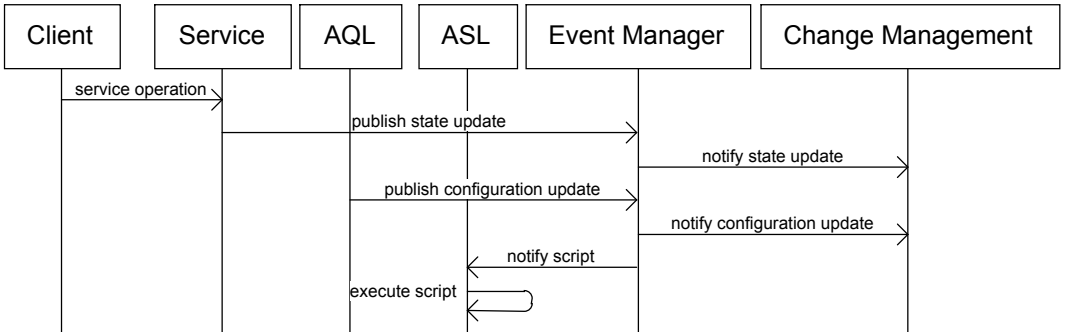

Figure 3.8: Component & Connector View of Component Control Layer

framework[2] for implementing OWL ontology support.

Continuing the thermometer example from Section 3.1, a plan may state which communication protocol a device should use in a given context for reasons of, e.g., power consumption, goodput, or throughput (Sperandio et al., 2008). As an example, an SWRL-based plan may state that a battery-operated device must use Bluetooth as a communication protocol if the battery level is below a certain threshold. Such an example is shown in Figure 3.2. In the example, a Thermometer that i) supports Bluetooth, ii) contains a battery with level less than 15%, and iii) is in the measuring state, should change its communication protocol to Bluetooth (for energy use reasons). Plans may easily conflict; consider, e.g., a requirement to optimize for energy use coupled with a requirement to optimize for throughput. This might, e.g., lead to plans simultaneously selecting Bluetooth and UDP for communication in our example. Resolving such conflicts is the responsibility of the Goal Management layer. We turn to this in Section 3.3.

In the example above, the execution of the plan is triggered by events from the Component Control layer on the state of the device and the battery level of that device. Publication of the device state is supported by state machine generation whereas the publication of battery level is an application-specific feature (unless it is connected to a state change). Such a sequence of events is shown in Figure 3.11. Finally, Figure 3.12 shows a deployment view of the Change Management layer of Hydra self-management, illustrating that SeMaPS is

[2] `http://protege.stanford.edu/`

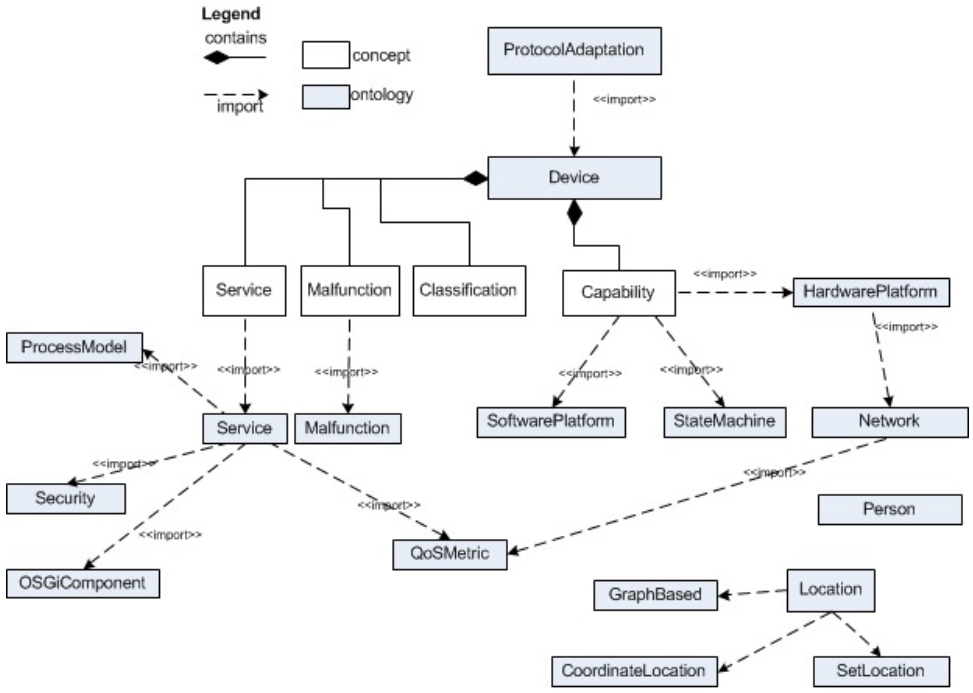

Figure 3.9: SeMaPS ontologies

$$
\begin{aligned}
& device : Thermometer(?device1) \\
& device : hasHardware(?device1, ?hardware) \quad \wedge \\
& hardware : supportProtocol(?hardware, ?protocol) \quad \wedge \\
& network : name(?protocol, ?name) \quad \wedge \\
& swrlb : containsIgnoreCase(?name, "Bluetooth") \quad \wedge \\
& hardware : primaryBattery(?hardware, ?bat) \quad \wedge \\
& hardware : batteryLevel(?bat, ?level) \quad \wedge \\
& swrlb : lessThan(?level, 0.15) \quad \wedge \\
& device : hasStateMachine(?device1, ?statemachine) \quad \wedge \\
& stateMachine : hasStates(?statemachine, ?state) \quad \wedge \\
& abox : hasURI(?state, ?uri) \quad \wedge \\
& swrlb : containsIgnoreCase(?uri, "measuring") \quad \wedge \\
& stateMachine : isCurrent(?state, ?cur) \quad \wedge \\
& swrlb : equal(?cur, "true") \\
\rightarrow\ & sqwrl : select(?device1, ?level) \wedge device : currentCommunicationProtocol(?device1, "Bluetooth")
\end{aligned}
$$

Figure 3.10: SWRL plan example

deployed on the same node as the reasoner component.

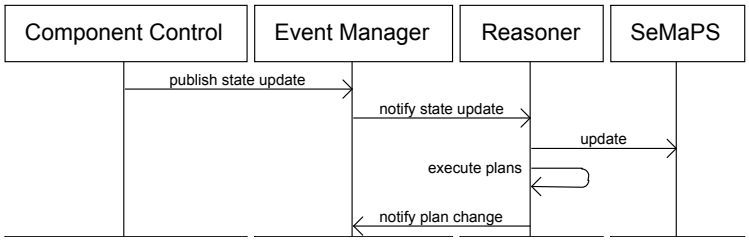

Figure 3.11: Component & Connector View of Change Management Layer

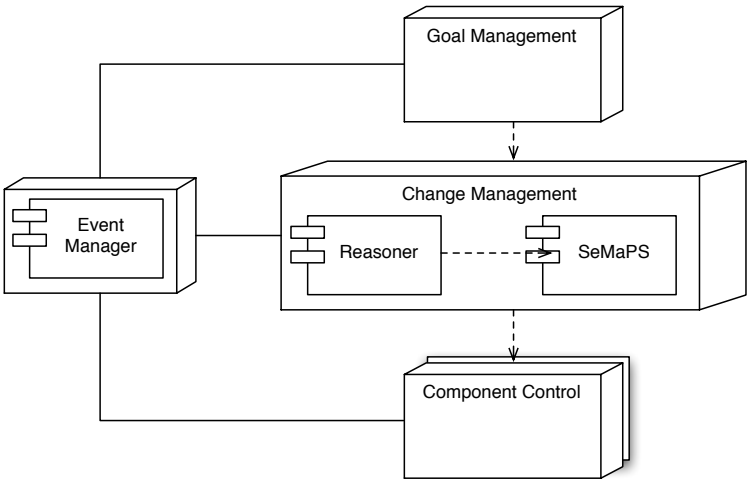

Figure 3.12: Deployment View of Change Management Layer

3.3 Goal Management

Goal Management is concerned with producing plans for how to reach a goal state given a high-level goal and a current system state. As such it is the most computationally intensive layer within the three-layer style.

In our realisation, the Goal Management layer consists of two main components (see Figure 3.17 for a typical deployment):

1. An *Optimizer* that will find a goal system configuration given a set of optimization constraints
2. A *Planner* that will find a plan (consisting of ASL scripts) for going from a current system configuration to a goal system configuration

Both components rely on having an up-to-date view of the system configuration. This is achieved by querying SeMaPS for system state and by listening to events published by the AQL sensors.

The query in Figure 3.13 illustrates the use of AQL. It provides the author means to select which parts of the query should be evaluated locally, and which parts globally. A Global modifier cannot be contained within a Local modifier. The data source, the AQL tables residing locally on each device, are specified in the innermost pairs of parentheses. The tables in this case are `ClientEndpoints` and `ServerEndpoints`. The `ClientEndpoints` table contains the

```
1  GLOBAL (
2      NATURALJOIN (
3          LOCAL (
4              RENAME [TargetEndpoint−>Address DeviceID−>ClientDID]( ClientEndpoints )
5          )
6          LOCAL (
7              RENAME [ServingEndpoint−>Address DeviceID−>ServerDID]
8  (ServerEndpoints)
9          )
10     )
11 )
```

Figure 3.13: An AQL Query

columns named (`DeviceID`, `ClientID`, `TargetEndpoint`, `Protocol`). It contains a tuple for each active client-connection, specifying the device id of the client (`DeviceID`) the local identifier of the client on that device (`ClientID`), the target endpoint for the service (E.g., for TCP it is the IP address and port number), and the protocol (`Protocol`). The `ServerEndpoints` contains the columns (`DeviceID`, `ServingEndpoint`, `Protocol`). `ServingEndpoint` denotes the endpoints on the server on which the service is reachable (for TCP/UDP, that is the IP address and port number the server socket is bound to). There is one tuple for each protocol/endpoint combination at which a service is available on this server. In this query, the tables with endpoints for clients and servers are each prepared locally, renaming the columns `TargetEndpoint` and `ServingEndpoint`, respectively, to `Address`, and renaming the device id columns to `ClientDID` and `ServerDID`. The rename is done to allow the convenience of using a natural join. The result of the local step is thus, conceptually at least, two tables with columns named (`ClientDID`, `ClientID`, `Address`, `Protocol`) and (`ServerDID`, `Address`, `Protocol`).The natural join of these yields a new table (`ClientDID`, `ServerDID`, `ClientID`, `Address`, `Protocol`) specifying the details of each active channel in the current system configuration.

The Optimizer uses genetic algorithms to find a solution given a set of constraints (Zhang et al., 2009b). The formulation of optimization problems are application-specific, but an example in the thermometer case would be a multi-objective optimization trading of energy consumption with throughput, the fitness function in the genetic algorithm being based on specific choices of protocols between service providers and consumers. Our implementation is based on the JMetal framework[3].

Given a current configuration and a goal configuration (as provided by the Optimizer), the Planner will attempt to construct an ASL script that transforms the current configuration into the given goal configuration.

Planning problems are described in the Planning Domain Definition Language (PDDL) (Mcdermott, 2000). A planning problem in PDDL consists of a domain description and a problem description. The domain description in a self-managed deployment consists of two parts that describe the operations that may be applied to architectural operations and as such defines the semantics of ASL operations:

- A *generic* part with ASL operations that apply to all applications. This includes operations such as deploying, starting, stopping, updating, undeploying components (Ingstrup and Hansen, 2009)
- An *application-specific* part with ASL operations that have specialized semantics for a domain. In our protocol change example, a `set_property` operation has been defined

[3] `http://jmetal.sourceforge.net`

that when set for a client of a service will cause that client to subsequently use the protocol that the property points to. This is further described in Chapter 4

Figure 3.14 shows the definition of the component undeploy operation in ASL: given a a device (`d`) and a component (`c`), the component may be undeployed from the device if it already exists at that device (as specified in the `:precondition`). The results (i.e., the `:effect`) is then that all packages that `c` provides are no longer available on the device, `d` (unless other components provide them).

```
(:action UNDEPLOY
    :parameters (?d ?c)
    :precondition (and (Component ?c) (Device ?d) (initiated ?d)
                       (At ?d ?c))
    :effect (and (not (At ?d ?c))
                 (forall (?p)
                         (when (and (Provides ?c ?p)
                                    (not (exists (?cm)(and (Component ?cm)
                                                           (Provides ?cm ?p)
                                                           (not (= ?c ?cm))
                                                    )
                                         )
                                    )
                               )
                               (not (AvailableAt ?d ?p))
                         )
                 )
            )
)
```

Figure 3.14: PDDL definition of the “undeploy” ASL operation

Planning problems are also described in PDDL and are defined by the goal that the Optimizer has provided. A simple excerpt is shown in Figure 3.15. As a result of solving the

```
(define (problem pb1)
  (:domain deployment)
  (:objects Cs Cc Ctcp Cudp Ds Dc ...)

  (:init
    ;; types
    (Component Cs) (Component Cc) (Component Ctcp) (Component Cudp)
    (Device Ds)(Device Dc)
    ...
    ;; relations
     (At Ds Cs)(At Dc Cc)(At Ds Ctcp)
    ...
    ;;states
     (initiated Dc)(initiated Ds)
    ...
  (:goal
     (PropertyValue Dc propertykey udptype)
  )
)
```

Figure 3.15: PDDL definition of planning problem

planning problem, on the client device (`Dc`), the following script will need to be executed:

```
set_property(propertykey, udptype);
```

And correspondingly, on the server device (`Ds`), the following script installing a UDP protocol component (`Cudp`) and starting a corresponding service (`sudp`) will need to be executed:

```
1 deploy_component(Cudp);
2 init_service(Cudp, Sudp);
3 start_service(Sudp);
```

These scripts are then communicatied to lower layers, using the Event Manager. This behaviour is illustrated in Figure 3.16 and a deployment is shown in Figure 3.17.

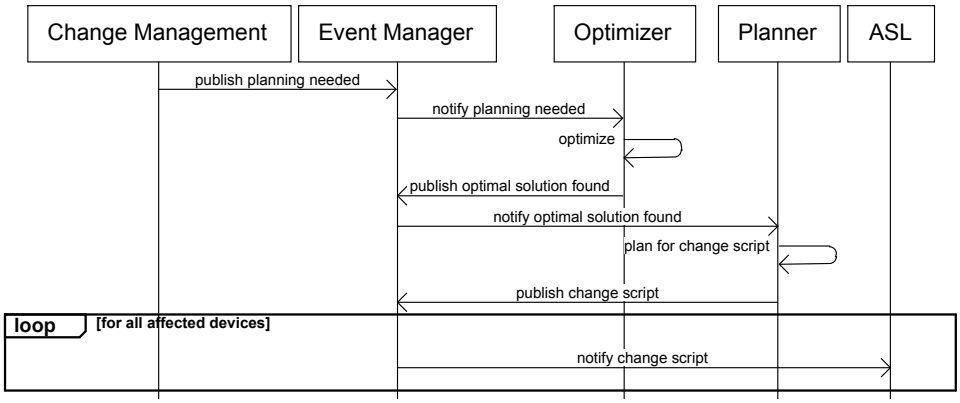

Figure 3.16: Component & Connector View of Goal Management Layer

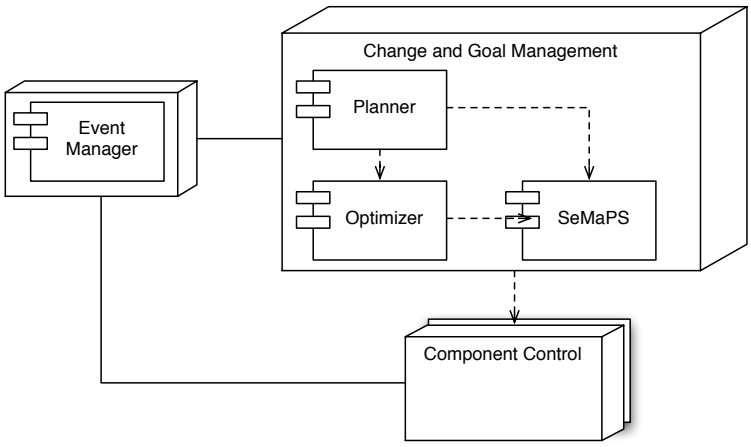

Figure 3.17: Deployment View of Goal Management Layer

4 Implementing a Self-Managed Application

In this chapter, we are going to use a simple scenario (already partially described in previous chapters) to explain how to build a self-managed application. The application is an environment monitoring application consisting of a central monitoring node connected to a number of thermometer services that provide temperature data. The intended deployment in the scenario is shown in Figure 4.1.

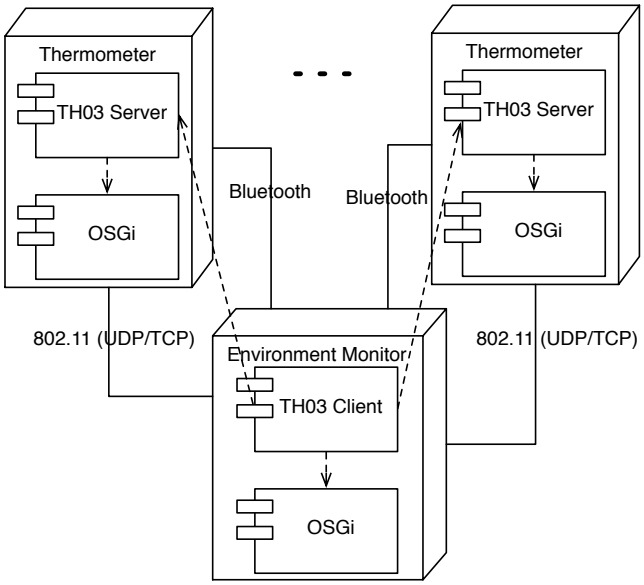

Figure 4.1: Deployment in the TH03 scenario

The Environment Monitor uses a number of Thermometers to perform its function. The TH03 Client deployed on the Environment Monitor communicates with a number of TH03 Servers on Thermometers through SOAP. Both the client and the servers are OSGi bundles deployed in an OSGi container. The connections between the client and servers can be both IEEE 802.11-based (giving high speed and throughput) and Bluetooth-based (giving longer battery life). These bundles are intended to run in the Component Control layer while we are interesting in managing the type of connection used between the client and servers.

Assuming this intended deployment, the steps involved in developing the application are:

1. For the Component Control layer, create the base application. Prepare the application for being controlled if necessary
2. For the Change Management layer, extend the SeMaPS ontologies to model the application concepts and plans
3. For the Goal Management layer, create application-specific optimization problems and extend the ASL domain descriptions to cover the application if necessary

In the following, we describe what needs to be done for each layer in more detail.

4.1 Implementing the Component Control Layer

The TH03 Server is defined using a WSDL file (as described in Section 3.1) and generated via the Service Compiler. The application developer furthermore needs to develop the actual functionality, in the TH03 case extracting temperature data to be returned upon request. To facilitate changing server protocol at runtime, the developer may use a set of Service Compiler runtime components that replace a standard OSGi HTTP Service implementation, enabling server protocols to be changed by changing bindings to protocol services. Finally, the developer needs to publish energy level events using the Event Manager (which may be accessed by a bridge from the OSGi Event Admin to the Hydra Event Manager).

The TH03 Client is also generated from the WSDL description and the developer needs to implement the behaviour of the Environment Monitor, including using the right protocol when requesting temperatures. The client protocol change logic is supported by framework classes generated by the Service Compiler.

4.2 Implementing the Change Management Layer

To enable reasoning on devices and services from the Energy Monitoring application, the SeMaPS ontologies are updated with concepts and rules for plans (see Section 3.2).

The change management layer needs to react to battery level change events from the Component Control layer. To do this a Protocol Reasoner bundle is implemented and deployed (see Figure 4.2). The Protocol Reasoner subscribes to battery level change events, applies level changes to the SeMaPS ontologies and decides the action to take (e.g., re-planning) after plans have been executed.

4.3 Implementing the Goal Management Layer

To enable application-specific optimization in the Goal Management layer, a Protocol Optimizer bundle is implemented. The bundle reads the necessary information from SeMaPS (in this case this includes devices and the current protocol), constructs an optimization problem class (based on a subclass of JMetal's Problem), and handles optimization results (from JMetal optimization).

To define a JMetal problem, a JMetal solution needs to be described. In our application, this is a vector of protocol choices per device-device connection involved, and protocol choice are evaluated based on latency and energy use of a protocol. The problem definition is then used as the basis for multi-objective optimization by JMetal.

Finally, the planning domain (for defining ASL operations) may need to be extended. In our case, this involves defining operations for setting protocol properties, resulting in the installation/activation of protocol bundles (see Section 3.3).

The final deployment is shown in Figure 4.2.

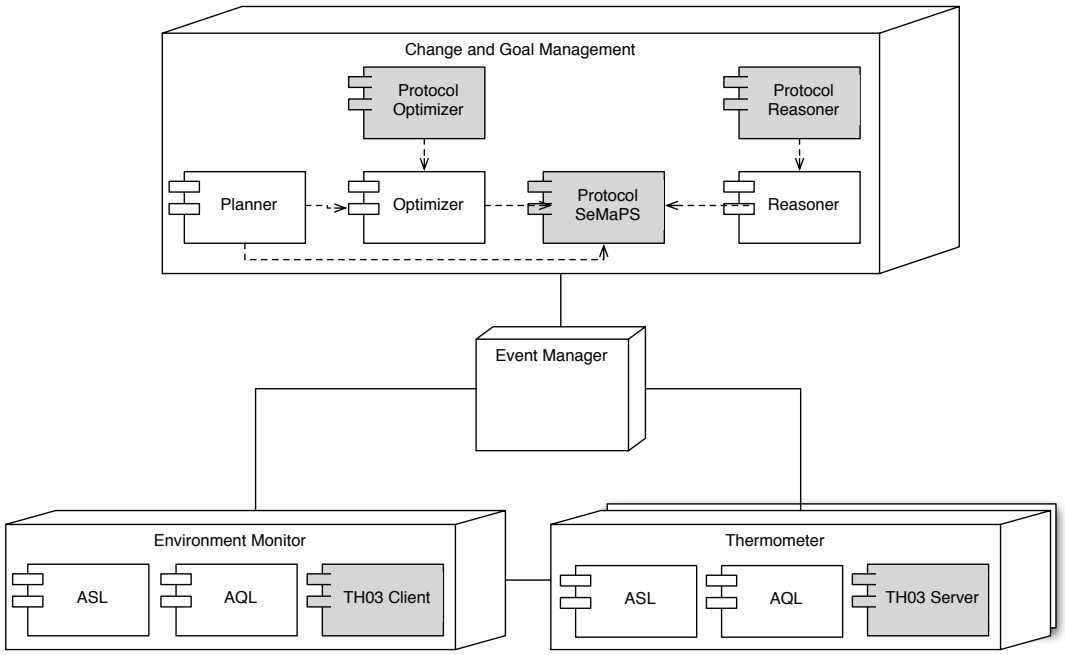

Figure 4.2: Final TH03 deployment. Gray components are tailored/specific to the application

5 Evaluation

This section presents an evaluation of the architecture and implementation with respect to completeness/utility and performance. First, we have realized a concrete family of scenarios concerned with optimizing the choice of protocols used in a system. The previous sections of this deliverable give example of this. Secondly, we have investigated that our approach can perform adequately by reporting measurements of key-tasks for individual components, and for specific technical scenarios covering several components in the architecture.

5.1 Ontology-Related Performance

As the Semantic Web and its supporting tools and technologies are still in the infancy stage, it is critical to make sure that we can get acceptable performance in a middleware system. In this process, a large number of tests and several strategies are adopted.

- Loading ontologies during components initialization and activation as ontologies loading into memory is a slow process
- Initializing static contexts by executing ontology queries in advance
- SWRL rule grouping

We need to investigate whether the complexity of underlying ontologies have big consequences with respect to performance for ontology update and reasoning (including SWRL inferring and SQWRL query). The complexity of an ontology is measured by several metrics in terms of defined classes, object properties, data type properties, restrictions (including cardinality, existential, etc.), total size (bytes) of an ontology or an ontology set, and rules. Please be noted that usually the update of an ontology property takes only a few milliseconds. As we are using the observer pattern to observe ontology property changes, the inferring of rules will happen immediately after the updates, and the update time will be more or less the same as the inferring time.

For the performance measurements, the following software platform is used: JVM 1.6.20-b02, Windows 7 64 bit, the hardware platform is: Thinkpad W700 T9400 2.53G CPU, 7200rpm hard disk, 4G DDR2 RAM. The time measurements are in millisecond. We use the default java heap memory.

Table 5.2 shows the performance effect of complexities of ontologies. We can see that there is little performance overhead when the complexity of ontologies increases. For example, the number of classes increases almost 8 times, object properties, and data type properties increase around 2 times, but the average time for inferring a rule varies very little. Therefore we tend to load the complete SeMaPS ontologies into memory when the self-management component starts. And from our former experiments, the loading of ontologies takes more than 3 seconds.

Table 5.2 shows the performance of rule inferring using the rule grouping feature where only a subset of rules are inferred, using the second ontology setting as in Table 5.2. We can see that the performance is better (average 3173.4 VS. 2057.77) with around 50% improvement.

Table 5.3 shows the performance of the optimizer, with maximum evaluations of 2000, population size 64, and cross over probability as 1. We can see on average it takes around 1.5 seconds to find optimal solutions.

Classes	object properties	data type properties	restrictions	total size	Time1	Time2	Time3	Time4	Time5	Average
73	39	58	46	105745	1841	3607	3482	2643	2413	2797.2
					2225	4140	3393	3621	3482	3372.2
					2588	4261	3348	2967	3524	3337.6
					1910	3194	3405	3394	2964	2973.4
					2167	3781	2691	2991	3631	3052.2
606	89	105	154	326,233	2264	3668	3422	3942	3336	3326.4
					1707	2653	2653	4555	1228	2559.2
					1678	3096	2934	3319	3807	2966.8
					2365	4396	4202	3414	4022	3679.8
					2352	3906	3890	3068	3459	3335
609	106	154	89	343,494	2184	2214	1326	2240	2526	2098
					2137	2902	2153	2685	2341	2443.6
					2431	2268	2371	1300	3550	2384
					1544	2372	2341	2528	2389	2234.8
					1662	2829	1493	2454	2337.2	

Table 5.1: Performance consequences of ontology complexity

1778	2090	1420	
1685	2465	2124	
3090	1530	1733	
2059	1919	2544	
1560	2184	2396	
1404	1638	1789	
1778	1702	2923	
1513	1544	1747	
2231	1155	2461	
1654	1935	1933	
1749	1778	2175	
2242	2543	1874	
1951	2480	2653	
1889	1997	2873	
2606	1825	2601	
1817	1856	2091	
1684	1950	1881	
1965	3182	1903	
2193	1794	2318	
1279	1684	1950	
1314	1561	2360	
2052	1701	2225	
1754	1529	2150	
1790	1856	1894	
1779	1374	1994	
2528	1202	2720	
2044	2029	2772	
2154	2560	2030	
1780	1654	3183	
1966	1779	1871	
1327	1373	2400	
1701	1592	1872	
1827	1639	1686	
2107	1561	2014	
1795	1998	3332	
1624	1561	2331	
1826	1498	1877	
2627	2076	2617	
2154	1780	2152	
2793	1810	2029	
2216	2653	2146	
1861	1499	2128	
2263	2154	3018	
2230	2247	1950	
2162	2435	2104	
2247	1903	1732	
1889	1764	1713	
1545	3137	2528	
1873	1934	2275	
1842	1811	2199	
2246	1857	2684	
2570	2279	2742	
1873	2496	2901	
3190	1826	3181	
2587	2387	1904	
1993.873	1923.018	2256.418	2057.77

Table 5.2: Performance after using rule grouping

952	1654	1201
1277	999	827
873	1092	421
1545	2029	671
1452	1498	1232
1081	1014	1175
858	828	905
1373	826	842
1295	1513	1482
1397	1109	1702
1318	905	718
1078	1062	1457
1498	952	1513
1680	967	1374
1039	1796	733
1264	1045	1467
891	951	1545
1451	1936	1450
968	468	889
796	2184	1466
999	1889	1326
1436	983	1488
1782	952	655
1171	1873	1757
1014	2216	1211
1313	2029	808
1069	1857	983
1108	2200	1654
477	1140	1794
1371	2264	1879
999	2092	1006
1451	2420	1061
1908	2327	1724
1642	1967	1202
1299	1561	1763
959	1530	1248
1162	1983	1873
1606	1562	1553
2622	1859	2056
2999	1656	1940
2987	1108	1357
2316	1639	1169
2013	2107	2215
2123	2341	1673
1974	1202	1343
2319	1577	1123
2019	1187	828
1694	1140	1171
1484	1562	1045
1896	531	1506
1983	1561	1413
1656	1296	1308
1497	1109	1486
1613	1077	1597
1628	1218	1826
1363	1873	1217
1048	921	1364
2439	1656	1427
2298	1014	1165
2392	1311	1190
2361	1343	906
967	1031	702
1447	1280	501
1447	1312	437
1275	764	780
1652	874	806
1518.646	1488.055	1329.291

Table 5.3: Optimizer Performance

5.2 Component-Related Performance

We measured performance on individual sub-components of our self-management system to locate potential bottlenecks and enable computation of their combined performance. An optimization cycle is adequate for validation as it involves all components in the self-management system.

The scenario starts when the reasoner detects a need for reconfiguration, and it involves the following components executing the following steps:

1. *Sensing*. AQL is used to retrieve the current system configuration. This is fed through the Event Manager to the Optimizer
2. *Optimization*. The Optimizer solves the optimization problem given by the state of the system and the current fitness function that reflects the current qualitative preferences for the system
3. *Planning*. The result of optimization is a desired target configuration. The target configuration, along with the current system configuration from step 1 constitutes a planning problem
4. *Actuation*. The result of planning is an ASL script transforming the system from its current configuration to the goal configuration. The result of planning is published by the planner through the eventmanager, and received by the ASL interpreter which executes it

The combined execution time for an optimization cycle is thus:

$$T_{total} = T_{sense} + T_{reason} + T_{optimize} + T_{plan} + T_{actuate} + 3 \cdot T_{event}$$

Where the $3 \cdot T_{event}$ is the time required to forward events among the components in between steps 1 to 4. Table 5.4 lists the measured values for each of the components in the above formula.

Parameter	Value (ms)	Measurements	Devices
T_{sense}	66.9	100	4
T_{reason}	2058	55	–
$T_{optimize}$	1445	5	–
T_{plan}	7	20	–
$T_{actuate}$	100	20	–
T_{event}	20.6	100	4

Table 5.4: Component Performance

Using the numbers in the table yields a total time of 3739 ms, which is within the acceptable limit for interactive applications. It is clear from the results that improvement to this number can only be achieved by reducing the time needed for reasoning and optimization, as the time consumption of the other sub-components of our self-management approach are insignificant in comparison.

The numbers confirm the soundness of choosing the three layer architecture for self-management. A key rationale in this architecture is that efficient algorithms should reside in lower layers. This is indeed the case as the components residing in the component control

layer (sensing, actuation) performs significantly faster than the components in the above layer.

The performance of the planning component appears good, however it should be noted that planning problems are inherently complex, so it is to be expected that when the planning problem instances increase in size, the performance of the planner will deteriorate significantly. Results of previous experiments show that the planning problems have to reach a size of more 70 architectural entities in the configurations before planning time exceeds 2000 ms. This confirms the allocation of the planning component to the top-layer, the goal management layer, along with the optimizer.

6 Related Work

Providing self-management features in pervasive computing environments is very attractive to make pervasive systems more usable and practical. A variety of technologies have been proposed for this purpose. CARISMA middleware (Capra et al., 2003) explored to use auction algorithms to dynamically resolve policy conflicts during context changes. Poladian et al (Poladian et al., 2006) used an analytical approach similar to combinatorial auctions in the project Aura to adaptively support everyday tasks of users. We are also working on including this in the self-management planning layer, as auction algorithms can find local optimal if required. But in general, from our experiences of preliminary tests, we will still use genetic algorithms as the main approach for finding global optimized solutions as it can find better quality of solutions with reasonable performance than the auction algorithms.

Genetic algorithms are being used in self-managing systems in which configurations are encoded as utility functions and where the problem of finding a (Pareto) optimal configuration becomes a multi-objective optimization problem (Hassan et al., 2008; Rouvoy et al., 2008). A recent approach within pervasive computing is the MUSIC middleware (Rouvoy et al., 2009). In MUSIC, components have QoS characteristics and the assumption is that QoS of a composition can be computed. In contrast to our work, MUSIC focuses on external Service Level Agreements (SLAs) and the fulfillment of those. Furthermore, it is not clear whether the proposed design is implemented and how planning will be realized. Besides the GA optimization engine, an IPP planner is also implemented to resolve the problem on how to achieve the optimal configuration in our goal management layer.

In Curry and Grace's work (Curry and Grace, 2008), The Model-View-Controller design pattern is adopted to improve concerns separation by encapsulating state, analysis, and realization operations. This MVC pattern improves flexibility, customization, and portability for self-representation of an autonomic system. We have actually adopted this MVC pattern in our design and implementation: the models are implemented using OWL ontologies, the controller parts are the corresponding model manipulation Java classes, the views are the various self-management schemas encoded with SWRL that are triggered by the updating of OWL models. This pattern is reinforced in our case by a Repository architecture style, and a layered architecture style.

The three-layered model was proposed by Kramer and Magee (Kramer and Magee, 2007), and there was no actual implementation details in (Kramer and Magee, 2007). A followed work is presented in (Sykes et al., 2008). Goals expressed in a temporal logic are used to generate reactive plans, and then configuration of domain-specific software components are configured based on the plan. In our case, we used a convenient IPP planner to generate the reaction plans to configure software components. We will explore more on the usage of state machine-based self-management based on our initial work (Zhang and Hansen, 2008b) and work in (Mostarda et al., 2010). In the future, we are considering to add an approach based on formal methods to regulate self-management tasks (Zhang et al., 2009a).

7 Conclusions

In this deliverable, we presented the WP4 embedded AmI architecture with focus on ontology- and context-enabled self-management. The architecture follows a three-layered approach in which computationally intensive self-management processes reside in upper layers (the Change Management and Goal Management layers) and in which processes on the lowest layer (the Component Control layer) may run on embedded devices.

Component Control is enabled by a combination of the Hydra Service Compiler, the Architectural Scripting Language (ASL) and the Architectural Query Language (AQL), the two former of which have been developed in Hydra. Change Management is enabled by a comprehensive set of OWL ontologies, SeMaPS, and related SWRL- and SQWRL-based reasoning. Change Management is enabled by a combination of a genetic algorithm-based optimizer and an ADL-based AI planner that generates ASL scripts.

Finally, the approach has been successfully validated for utility and performance through a set of protocol change-related scenarios.

Bibliography

Brooks, R. A. (1991). Intelligence without representation. *Artif. Intell.*, 47(1-3):139–159.

Capra, L., Emmerich, W., and Mascolo, C. (2003). CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, pages 929–945.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. (2003). *Documenting software architectures: views and beyond*. Addison-Wesley, Boston.

Curry, E. and Grace, P. (2008). Flexible Self-Management Using the Model-View-Controller Pattern. *IEEE software*, 25(3):84–90.

Diao, Y., Hellerstein, J. L., Parekh, S., Griffith, R., Kaiser, G. E., and Phung, D. (2005). A control theory foundation for self-managing computing systems. *Selected Areas in Communications, IEEE Journal on*, 23(12):2213–2222.

Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., and Zambonelli, F. (2006). A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1(2):223–259.

Dolog, P. (2004). Model-driven navigation design for semantic web applications with the uml-guide. *Engineering Advanced Web Applications, In Maristella Matera and Sara Comai (eds.)*.

Erl, T. (2005). *Service-oriented architecture: concepts, technology, and design*. Prentice Hall PTR Upper Saddle River, NJ, USA.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131.

Fielding, R. T. and Taylor, R. N. (2002). Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150.

Garlan, D., Cheng, S. W., Huang, A. C., Schmerl, B., and Steenkiste, P. (2004). Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54.

Gat, E. (1998). On three-layer architectures. *Artificial Intelligence and Mobile Robots*, pages 195–210.

Hansen, K., Zhang, W., and Ingstrup, M. (2008a). Towards Self-Managed Executable Petri Nets. In *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, 2008. SASO'08*, pages 287–296.

Hansen, K. M. and Ingstrup, M. (2010). Modeling and analyzing architectural change with alloy. In Shin, S. Y., Ossowski, S., Schumacher, M., Palakal, M. J., and Hung, C.-C., editors, *SAC*, pages 2257–2264. ACM.

Hansen, K. M., Zhang, W., and Fernandes, J. (2008b). OSGi based and Ontology-Enabled Generation of Pervasive Web Services. In *15th Asia-Pacific Software Engineering Conference*, pages 135–142, Beijing, China.

Hansen, K. M., Zhang, W., and Soares, G. (2008c). Ontology-enabled generation of embedded web services. In *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering*, pages 345–350, Redwood City, San Francisco Bay, USA.

Hassan, O., Ramaswamy, L., Miller, J., Rasheed, K., and Canfield, E. (2008). Replication in Overlay Networks: A Multi-objective Optimization Approach. In *4th International Conference on Collaborative Computing: Networking, Applications and Worksharing (COLLABORATECOM-2008)*, Orlando, FL, USA.

Ingstrup, M. and Hansen, K. M. (2005). A declarative approach to architectural reflection. In *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*, pages 149–158.

Ingstrup, M. and Hansen, K. M. (2009). Modeling architectural change: Architectural scripting and its applications to reconfiguration. In *WICSA/ECSA*, pages 337–340. IEEE.

Kramer, J. and Magee, J. (2007). Self-Managed Systems: an Architectural Challenge. *International Conference on Software Engineering*, pages 259–268.

Kriens, P. (Ed) (2005). OSGi Service Platform Core Specification. Release 4. The OSGi Alliance.

Labella, T. H., Dorigo, M., and Deneubourg, J.-L. (2006). Division of labor in a group of robots inspired by ants' foraging behavior. *ACM Trans. Auton. Adapt. Syst.*, 1(1):4–25.

Mcdermott, D. (2000). The 1998 ai planning systems competition. *AI Magazine*, 21(2):35–55.

Mostarda, L., Sykes, D., and Dulay, N. (2010). A State Machine-Based Approach for Reliable Adaptive Distributed Systems. In *2010 Seventh IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems*, pages 91–100. IEEE.

Ottino, J. M. (2004). Engineering complex systems. *Nature*, 427(6973):399.

Poladian, V., Sousa, J., Garlan, D., Schmerl, B., and Shaw, M. (2006). Task-based adaptation for ubiquitous computing. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, Special Issue on Engineering Autonomic Systems*, 36(3).

Ranganathan, A. and Campbell, R. H. (2004). Autonomic pervasive computing based on planning. In *Proceedings of International Conference on Autonomic Computing.*, pages 80–87.

Rao, A. and Georgeff, M. (1995). BDI agents: From theory to practice. In *Proceedings of the first international conference on multi-agent systems (ICMAS-95)*, pages 312–319. San Francisco.

Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S. O., Lorenzo, J., Mamelli, A., and Scholz, U. (2009). Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments. In Cheng, B. H. C., de Lemos, R., Giese, H., Inverardi, P., and Magee, J., editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 164–182. Springer.

Rouvoy, R., Eliassen, F., Floch, J., Hallsteinsen, S., and Stav, E. (2008). Composing Components and Services Using a Planning-Based Adaptation Middleware. *LECTURE NOTES IN COMPUTER SCIENCE*, 4954:52–67.

Sperandio, P., Bublitz, S., and Fernandes, J. (2008). Wireless Device Discovery and Testing Environment. Technical Report D5.9, Hydra Consortium. IST 2005-034891.

Sykes, D., Heaven, W., Magee, J., and Kramer, J. (2008). From goals to components: a combined approach to self-management. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 1–8. ACM.

Zhang, J., Goldsby, H., and Cheng, B. (2009a). Modular verification of dynamically adaptive systems. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 161–172. ACM New York, NY, USA.

Zhang, W. and Hansen, K. (2008a). Semantic web based self-management for a pervasive service middleware. In *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, 2008. SASO'08*, pages 245–254.

Zhang, W. and Hansen, K. (2009). An Evaluation of the NSGA-II and MOCell Genetic Algorithms for Self-management Planning in a Pervasive Service Middleware. In *14th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2009)*. IEEE Computer Society Washington, DC, USA. 192-201.

Zhang, W. and Hansen, K. M. (2008b). An OWL/SWRL based Diagnosis Approach in a Web Service-based Middleware for Embedded and Networked Systems. In *The 20th International Conference on Software Engineering and Knowledge Engineering*, pages 893–898, Redwood City, San Francisco Bay, USA.

Zhang, W. and Hansen, K. M. (2008c). Towards self-managened pervasive middleware using OWL/SWRL ontologies. In *HCP-2008 Proceedings, Part II, MRC 2008 – Fifth International Workshop on Modelling and Reasoning in Context*, pages 1–12.

Zhang, W., Schutte, J., Ingstrup, M., and Hansen, K. M. (November 24-27 2009b). Towards optimized self-protection in a pervasive service middleware. In *7th International Conference on Service Oriented Computing, Joint ICSOC&ServiceWave 2009 Conference*, pages 404–419, Stockholm, Sweden). Springer LNCS 5900.