Contract No. IST 2005-034891

Hydra

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

D4.8 Self-* properties DDK prototype and report

Integrated Project
SO 2.5.3 Embedded systems

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

Published by the Hydra Consortium **29 December, 2008- version 2.0**
Lead Contractor: Fraunhofer FIT

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

Dissemination Level: Public

Document File:	D4.8 Self-* properties DDK prototype and report
Work Package:	WP4
Task:	T4.3
Document Owner:	Weishan Zhang (UAAR), Mads Ingstrup (UAAR)

Document history:

Version	Authors	Date	Changes Made
0.1	Mads Ingstrup (UAAR)	2008-10-06	Outline
0.2	Mads Ingstrup	2008-11-03	Added scenario, started describing the hydra self management architecture.
0.3	Weishan Zhang (UAAR)	2008-11-16	added draft for semantic web based approach, scenarios
0.4	Mads Ingstrup	2008-11-18	added survey of related work
0.5	Weishan Zhang	2008-11-23	added planning survey, architectural semantic models
0.6	Weishan Zhang	2008-11-24	structure changes, planning survey, service filtering
0.7	Weishan Zhang	2008-11-27	structure changes to planning chapter, formulating self-management and utility functions
0.8	Weishan Zhang	2008-12-9	updates incorporate Klaus's comments (scenario, SeMaPS ontologies sections, etc.)
0.9	Weishan Zhang	2008-12-10	QoS ontology section
1.0	Weishan Zhang	2008-12-11	Justifying OWL-DL and SWRL with respect to review comments, executive summary updates, architectural rules
1.1	Mads Ingstrup	2008-12-13	Updated the survey to increase focus on Hydra; introduction; substantial update of the description of architectural scripting; new figures for the overall architecture.
1.2	Weishan Zhang and Mads Ingstrup	2008-12-13	structure changes, scenario changes, example for using utility functions
1.3	Mads Ingstrup	2008-12-14	First complete proof reading; checked spelling.
1.4	Weishan Zhang	2008-12-15	overview of the Hydra Self-* Architecture and tools/artifacts, structure changes, role of deliverable, proof reading changes.
1.5	Weishan Zhang and Mads Ingstrup	2008-12-15	conclusions, APIs usage, and future work.
1.6	Weishan Zhang	2008-12-15	change ASL implementation back to its design chapter. Ready for internal review
1.7	Weishan Zhang	2008-12-16	Deployment of self-management components with respect to review comments
1.8	Weishan Zhang	2008-12-17	added rule comparisons
1.9	Mads Ingstrup	2008-12-21	Changes cf comments from internal reviewers.
2.0	Weishan Zhang	2008-12-29	genetic algorithm section, proof reading, final for submission.

Internal review history:

Reviewed by	Date	Comments
Andreas Zimmermann (FIT)	2008-12-19	Approved with Comments
Tomas Sabol, Jan Hreno (TUK)	2008-12-19	Approved with Comments

Contents

List of Figures

List of Tables

Executive Summary

This deliverable documents the achievements of Hydra within the area of self management until month 30. The following tasks have been achieved compared to the last deliverable D4.3 (Ingstrup et al., 2008). They are listed in the order in which they are documented in this deliverable:

- Clarification of self-management capabilities running on different type of devices (D0 to D4) as classified in deliverable D5.4. (chapter 1)
- A more thorough account of related work: The increased page and reference count in the related work section is the result of a more detailed analysis of the relationship between different types of adaptation and detection algorithms. (chapter 2)
- Three new cases which takes up the challenge of self-healing, self-configuration and goal-management set out as targets for future work in deliverable 4.3 (Ingstrup et al., 2008, p. 15). (chapter 3)
- A clarification of the overall architecture for self-management in Hydra. This describes how the previous self-* prototypes have been integrated together with the new work, how the self-management components collaborate with other Hydra components, as well as frame our discussion of future work. (Chapter 4)
- The design and implementation of an architectural scripting language on top of OSGi, which is used both for self-management, and for supporting the development and test of systems in Hydra. (chapter 5)
- The initial specification of a testbed for self-management, including an implementation of the ANT configuration tool to include the architectural scripting language operations. (chapter 5)
- The realization of the three cases which exemplifies self-repair and self-configuration, how QoS can be used to achieve self-adaptation given a particular quality goal. (chapter 5,7, 9)
- An elaboration of the reasons for choosing OWL-DL[1] and SWRL[2] as the means for achieving ambient intelligence in Hydra, together with the discussion of their weakness. This is a reaction to review comments on why Hydra is choosing OWL-DL+SWRL for ambient intelligence. We also published this result in a SASO 2008 paper. (chapter 6)
- A formulation of the self-management for pervasive service (SeMaPS) ontology set, adding new ontologies (MessageProbe, QoS, Location, Time, Person, OSGiComponent, Connector, ArchitectureStyle) to SeMaPS ontologies. (chapter 6)

[1] http://www.w3.org/TR/owl-guide/

[2] SWRL: A Semantic Web Rule Language Combining OWL and RuleML. http://www.w3.org/Submission/SWRL/

- A set of self-management rules is developed, which covers self-adaptation, self-diagnosis, self-configuration, and QoS are considered in some of the rules. (chapter 7)
- The design of a solution for using probability in semantic ontologies based on the survey of current situations in handling probabilities in semantic web. (chapter 7 Section 7.5)
- Propose a self-management model and Hydra utility functions, based on the survey for planning mechanisms. Genetic algorithm is promising to serve as the planning mechanism in Hydra. (chapter 8)
- Five papers are published based on this task (SEKE 2008, SASO 2008 (two papers), MRC 2008 workshop, IOT&S workshop), as in the Appendix.

The following sub-tasks of T4.3 remains outstanding but should be finished by month 36, as according to the plan in the new DOW (V7.23):

- More planning algorithms need to be investigated, and implementation and evaluation of the planning algorithms.
- Self-protection, in collaboration with WP7. In (Ingstrup et al., 2008) self-protection was outlined as a point for future work. This is still not directly addressed with the work in this deliverable. The current view in wp4 is that this should be implemented in collaboration with wp7 because it is likely to involve security. With that said however, the work on self-management documented in this deliverable is generic, and should be as applicable to self-protection as to other branches of self-* functionality.
- Further enhancements to SeMaPS ontologies and rules development for scenarios arisen during the project goes, and more through evaluations for self-adaptation, self-configuration.
- Integrating the Semantic and CPN versions of Flamenco, probably will be assumed by the efforts on Goal management layer as it is more interesting as per the capabilities of self-management in Hydra.
- The testbed for self management must be specified in more detail, particularly by formalizing the procedure for adding support for new devices. Because only one device is currently supported, the addition of support for one or more additional platforms is needed to inform this work and verify the current design. In addition to this, the specification of interfaces for pluggable self-* components is needed, along with adaptation of the current prototypes to exemplify this.

The role of this deliverable in Hydra Self-management is a challenging research area for pervasive computing. This deliverable lays out a solid foundation for how we achieve self-management goals in Hydra, to achieve the vision of Ambient intelligence. The work is based on extensive survey on existing related work, and our former deliverable D4.3 (Ingstrup et al., 2008). Specifically, this deliverable serves for Hydra:

- Elaborating of a semantic web based self-management approach for Hydra, based on self-management ontologies SeMaPS, in which OWL-DL and SWRL are used.

- An extensive ontologies set SeMaPS (including a set of self-management rules covering full spectrum of self-management features) could be used by other work packages for context modeling and context reasoning, and other self-management purposes. SeMaPS is serving as knowledge base for self-management in Hydra.
- An approach for pervasive system test bed build up and related prototypes can be used for testing self-management features.
- An prototype platform for semantic web based self-management serves as foundation for the implementation of full self-management features, and approach for planning is proposed.

1 Introduction

This deliverable consists of a set of software prototype components for the Hydra middleware. The deliverable is a prototype and a report. For the prototype part we give a summary of the components that have been developed. The report details the research contributions in which these prototypes play a role.

The purpose of this deliverable is to present the research into self-managing techniques and how it has contributed to the Hydra middleware. In addition, this deliverable should work as a basic introduction for developers to start using the self-management components. More technical details on other components can be found in other deliverables: Limbo details are in Deliverable D4.2 (Hansen et al., 2007), and details on preliminary work on self-diagnosis with Flamenco including the agriculture scenarios used in Flamenco components are given in D4.3 (Ingstrup et al., 2008).Deliverable 12.5 (Fernandes et al., 2008) contains a tutorial of ASL-ANT.

1.1 Components Overview

The following components are provided as a result of the work documented in this deliverable.

- ASL/ANT: An implementation of architectural scripting which extends ANT so it can be used to set up distributed configurations of services.
- ASL/Self-*: A bundle for the Equinox OSGi platform that partially implements the component control layer of the 3L architecture adopted for Hydra self-management. It allows reconfigurations to be executed on this platform.
- SeMaPS ontologies, serve as the knowledge base for self-management in Hydra, in which both static and dynamic contexts are modeled, and complex contexts are modeled with SWRL.
- Self-management rules set, including new rules for self-diagnosis, self-adaptation, self-configuration.
- Updated Flamenco/SW (Semantic Web based self-management component) with enhanced capabilities for self-diagnosis, and rule grouping features are used to improve flexibility and performance.

1.2 Deployment of self-management components

In Hydra, devices are classified into 5 categories as in D5.4 (Sperandio et al., 2007), namely D0 to D4 devices as shown in Table 1.1. In the last column, we listed possible deployment of self-management components. Currently, our work is mainly concentrated on making the self-management component work on powerful node like the D3, D4 devices, which have almost unlimited resources for exploring self-management. In reality, if a device can run JVM (version 1.6), then there should be no problem for running self-management components. For some storage and CPU limited devices, they can partially running only the SWRL reasoning (using Jess[1]).

[1] http://herzberg.ca.sandia.gov/

Table 1.1: Devices classification in Hydra

Type	Description	Self* components
D0	Non-HED. Specific communication protocol (BT, ZigBee). Need of a proxy in D4	NA
D1	Non-HED. WS support (use of Limbo).	ASL/Self*, Partial or full rules set and SWRL reasoning
D2	HED. Specific communication protocol. Need of a bridge (dedicated or in D3-D4)	ASL/Self*, Partial or full rules set and SWRL reasoning
D3	HED. Bridge hosting	ASL/Self*, partial or full SeMaPS ontologies, Partial or full rules set and SWRL reasoning, partial of full Flamenco
D4	Gateways. Proxy and bridge hosting	ASL/Self*, full SeMaPS ontologies, full rule sets, full Flamenco

2 State of the art

In this chapter we consider the state of the art in self managing systems. Firstly we introduce the concept of self managing systems and describe the method used to survey existing work. The next section discusses the overall conceptual and architectural basis for autonomic computing. It works as a foundation based on which we can discuss concrete adaptation techniques. Next, in section 2.3,2.4, 2.5 we go through the qualities reliability, performance, and usability using each as a perspective from which to discuss autonomic techniques. Finally, in section 2.6 we conclude on the state of the art from the perspective of Hydra and draws out the implications that motivate the work presented in the remainder of this deliverable.

2.1 Introduction

A self managing, or autonomic, system is one in which technology is deployed specifically for the purpose of managing other technology (Dobson et al., 2006). There are several reasons why this is appealing. Firstly, as systems become more complex the effort required to manage them grows. Building a system specifically to ease this task and automate parts of it reduces the need for human intervention. Therefore it makes it practically possible to build more complex systems. The trend for the past couple of decades indicates that this is required.

Secondly, the dynamism characteristic of pervasive, ubiquitous and mobile computing makes it increasingly untenable to build systems that only work when a set of statically fixed assumptions are true. Thus a system must be able to adapt itself or be adapted to a change of circumstance.

We propose to cast the challenge of autonomic computing in terms of the qualitative requirements such as for reliability, security, usability etc. Focusing on the qualitative aspects of a system largely leaves out the functional aspects. This is deliberate, because detecting and accommodating a change in the functional requirements of a system is a considerably harder problem to automate. Building a system that does the right thing for a user is already hard to do for highly trained human developers, so automating it seems intuitively hard. Another and more pragmatic reason for leaving out techniques to accommodate changing functional requirements is that to our knowledge there is very little to survey in this area. The dynamism of functional requirements is a research problem addressed by the end-user development research community.

A key challenge in realizing an autonomic system is the detection of when the auxiliary managing logic should be activated and what it should do to handle a detected problem. We can understand this challenge in terms of the relationship between the requirements to a system on the one hand, and whether the system's operation conforms with these requirements on the other. The autonomic control logic should then be activated when there is a current or predicted discrepancy between desired and actual operation of the system.

Discrepancies between actual and desired operation can arise because users' requirements to the system may change, or because changes in the environment may mean that another configuration of the system is needed to meet a stable set of requirements. Depending on the design philosophy used in a system, a change in the environment, the conditions under which a system provides its functionality, will often manifest itself as a fault or error in the system. Therefore the avoidance of or recovery from errors is the goal of a particularly important class of autonomic techniques, which help to further improve reliability and

availability of a system.

2.1.1 Related Work

Other researchers have surveyed areas related to autonomic computin (Sterritt et al., 2005; Dobson et al., 2006; Nami and Bertels, 2007; Elkhodary and Whittle, 2007). Sterrit et al. (Sterritt et al., 2005) gives a broad overview of the field. Reza and Bertels (Nami and Bertels, 2007) focus on the historical development and current challenges of the area. Dobson et al. (Dobson et al., 2006) provide a comprehensive survey of autonomic communications. Elkhodary and Whittle (Elkhodary and Whittle, 2007) survey approaches to adaptive security. In contrast to this earlier work, we focus on particular techniques for diagnosis and subsequent adaptation, and we do so from an architectural perspective.

2.1.2 Survey Method

In autonomic computing several techniques exist that aim to improve certain quality attributes of a system. We want to position these techniques and concerns relative to each other, and map points at which they are synergies and conflicts between them.

To enable comparison of techniques we must describe those of their properties that are relevant to assessing their compatibility. Two techniques are compatible if the assumptions they make do not conflict, and more so if they make the same assumptions. For instance, a technique assuming a service oriented architecture is arguably easier to combine with a second technique that makes the same assumption, and conversely, may be difficult to combine with one that is not service oriented, but instead relies on e.g. CORBA. The challenge of combining or reusing various techniques is closely related to that of making reusable software components.

Garlan et al. (Garlan et al., 1995) have shown that a crucial reason why reuse is often hard is that components made for reuse often make incompatible architectural assumptions. While our unit of analysis is in the first instance more general than a particular implemented component, we believe that architectural assumptions made by particular algorithms are equally important to their combination and reuse. Therefore our taxonomy is also concerned with the architectural assumptions made by each technique.

Software architecture is about the structure or structures of a system (Bass et al., 2003b) and how these affect its quality attributes. Our perspective in this survey is architectural because we classify the diagnosis and adaptation techniques according to the architectural assumptions they rely on, and the quality attributes they help improve. This approach helps elicit the information necessary to compare the different techniques, and in particular, it may aid in assessing how compatible two techniques are.

The inclusion critera for the techniques in our survey have been:

1. The soundness of the work, including, importantly, whether it has been evaluated.
2. The scope of the technique. If the technique applies only to a very narrow set of circumstances and are unlikely to be reused outside those circumstances, then it is of lesser importance to the readers of a survey.
3. The subset should be representative. Due to limits in space and the large body of work that exists, we cannot include every technique. However we have strived to keep the subset selected representative of state of the art. Thus many of the included techniques aims to improve reliability, because a disproportional number of techniques focus on this compared to other quality attributes.

4. The diversity of techniques. While the surveyed subset of techniques should be representative, it is also important to reflect in the survey the breadth of the techniques that exist. Reading a survey should be for inspiration as much as for reference.

A summary of the surveyed techniques and their properties is shown in table 2.1.

2.2 Conceptual and Architectural Basis of AC

Parashar and Hariri (Parashar and Hariri, 2005) describe four types of challenges for research in autonomic computing:*Conceptual* challenges concerning how we understand autonomic systems, including models and abstractions of them; the *Architectural* challenges of what architecture can enable self-management at various levels of granularity, locally or globally and so they can be specified, implemented and controlled in a predictable and robust manner; *Middleware* challenges about what core services are needed to support realization of autonomic systems subject to particular and perhaps varying quality requirements; and finally *Application* challenges that are concerned with the programming, development and maintenance of concrete autonomic applications.

These four distinct concerns of autonomic computing form a cascading categorization: systems that differ in their conceptual approach are likely to be more different than those within the same conceptual category but with different architectures; and systems within both the same conceptual and architectural categories but with different middleware designs are likely to be more similar still, and so on.

With respect to these four categories, our concern in this section is to lay out the conceptual and architectural basis for the subsequent discussion of more middleware and application related issues from the perspective of our three chosen qualities, reliability, performance and usability.

2.2.1 Conceptual perspective

There has been at least two main conceptual approaches to building autonomic systems. One is inspired by traditional AI with the explicit representation and interpretation of plans as a basis for action, e.g. the 3-layer architecture by (Kramer and Magee, 2007a). The other major conceptual approach is to build systems without any explicitly represented overall plan, e.g. inspired by the decentralized control in ant colonies (Labella et al., 2006), or by evolution as an adaptation mechanism (Goldsby et al., 2007). This is based on the observation that complex self-managing systems in nature, e.g. bodies made of organs, cells etc. are build this way and are known to work well(Babaoglu et al., 2006; Dobson et al., 2006). However it is difficult to engineer these kinds of systems(Ottino, 2004).

Although these two approaches are conceptually different it does not follow that they are mutually exclusive in a given system architecture. It is an open topic how these two approaches can be combined. However, the human nervous system that provide some of the inspiration for the vision of autonomic computing (Parashar and Hariri, 2005) encompasses both high level cognitive explicit/conscious planning as well as relying on lower level more emergent properties for self management and healing.

A third conceptual approach to autonomic systems is based on the system model used in control theory (Diao et al., 2005). The sub-taxonomy for conceptual approaches to self management is shown in figure 2.1.

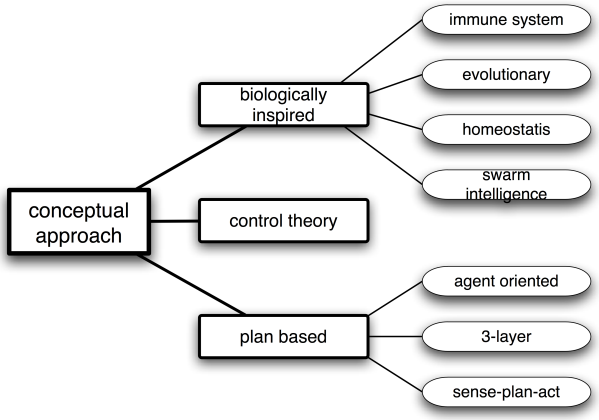

Figure 2.1: The major conceptual approaches to building self managing systems.

2.2.2 Architectural perspective

Kramer and Magee (Kramer and Magee, 2007a) recently proposed a reference model for self-managed systems based on an interpretation of Gat's three layer architecture for autonomous robotics (Gat, 1998a). This model for self-management of software systems contrasts earlier approaches based on Sense-Plan-Act architectures (Gat, 1998a) which have also been used in self-management systems (such as by (Garlan and Schmerl, 2002; Arshad et al., 2004; Garlan et al., 2004)).

Kramer and Magee argue that handling self-management on an architectural level is appropriate in terms of level of abstraction and generality (as does (Dashofy et al., 2002; Oreizy et al., 1999; Garlan and Schmerl, 2002; White et al., 2004; Garlan et al., 2004) and others) and casts Gat's three layer architecture in terms of conceptual layers of a self-management system, illustrated in figure 4.1:

Component Control Layer. This layer includes sensors, actuators, and simple control loops. In a self-managed system, this layer consists of elements that perform application functions ("control loops"), reporting of state to upper layers ("sensors") and facilities for creating, changing, and deleting elements ("actuators"). Techniques used at this layer includes context awareness (Dey et al., 2001; Gu et al., 2005; IST Amigo Project, 2006), reflection (Maes, 1987; Ingstrup and Hansen, 2005), architecture discovery (Schmerl et al., 2006) or other forms of monitoring (Snodgrass, 1988) that enable programmatic retrieval of data about the environment and the system itself. Such input is vital to adaptation and reconfiguration algorithms, both to detect when they should be initiated and to provide the sense-data they operate on.

Change Management Layer. Based on state reported from the Component Control Layer, the Change Management Layer executes precomputed plans and change the elements in the Component Control Layer. If a conditions is met for which a plan does not exist, a new plan may be requested from the Goal Management Layer.

Goal Management Layer. This layer creates new plans based on high-level objectives (e.g. a service level agreement for the system) of the running system. Often compute-intensive re-planning is done. Examples of work in this layer include (Srivastava et al.,

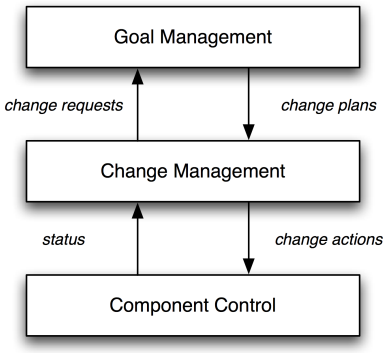

Figure 2.2: Three Layer Architecture Model for Self-Management

2004; Ranganathan and Campbell, 2004)

A conceptual model such as this leaves open how a particular set of deployed runtime entities relates to each layer. In an agent based architecture such as advocated by e.g. (Weyns and Holvoet, 2007), for instance, each agent could implement all layers and use a decentralized algorithm with no overall explicit plan for achieving autonomic properties at the systemic level. This also matches the model used in a component based framework proposed by Liu et al. (Liu et al., 2004), in which each (agent-like) component is rule based and specified through behaviour rules and interaction rules. In these two cases each component is itself self managing, a requirement also stated by (White et al., 2004).

In another interpretation, there may be a centralized planner implementing the goal management layer. Even with such a choice of runtime architecture the plan generation could be biologically inspired. For instance by letting plans be state-machines that, given a particular set of input events (sensed data) generate a particular stream of output events (actions), plans could be generated with some form of evolutionary method for plan selection such as in (Goldsby et al., 2007).

2.3 Reliability

Reliability is “the capability of the software product to maintain a specified level of performance when used under specified conditions” (iso, 2001). This includes fault tolerance and recoverability. Several techniques in AC helps further reliability in that they seek to either tolerate, avoid or recover from faults.

In the GAIA meta-operating system for pervasive computing (Chetan et al., 2005; Ranganathan et al., 2003; Roman et al., 2002) fault tolerance is achieved using a fail-stop fault model (Schneider, 1984). Each application sends out a heartbeat message, the absence of which indicates a fault that prompts GAIA to infer, based on context information, an appropriate device on which that application may be restarted (Chetan et al., 2005).

In several infrastructures for pervasive computing, user tasks or service compositions are explicitly represented and thus serve as the object of reconfiguration and monitoring for autonomic techniques (Sousa et al., 2006; Garlan et al., 2002; Amundsen and Eliassen, 2008).If a service required by a particular task becomes unavailable, an alternative is searched for

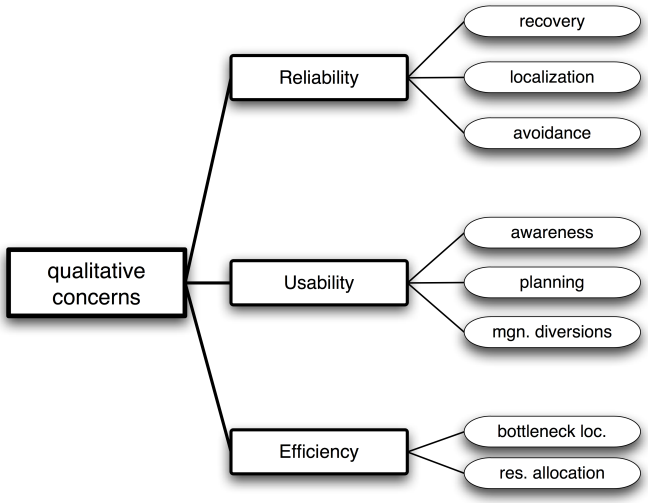

Figure 2.3: The quality attributes and strategies used to characterize each technique.

using service discovery, and chosen possibly subject to various constraints expressed in the composite specification.

Baresi et al. (Baresi et al., 2007; Baresi and Guinea, 2007) in a similar venue demonstrates a framework for making web service compositions adaptable. It provides monitoring facilities that can collect data from dedicated probes and analyze it against specified constraints. This way the dynamic selection of services at runtime can be guided by higher level constraints inferred based on e.g. context information. For instance, the choice of what service is used for placing telephone calls can be made based on a user's location. This way, a specified level of performance can be maintained under specified conditions, but leaving greater flexibility to specify those conditions.

Ahmed (Ahmed et al., 2007b,a) detects faults based on the rate of change in a sensed property. For instance if the rate of change of allocated memory in a given period is higher than a set threshold, an error is reported. However it is unclear what the accuracy of this heuristic is and which kinds of faults it is most appropriate for.

A number of other approaches for dealing with errors exist that have not been tried in a pervasive computing scenario, but which nevertheless may inspire future work in this area. Kiciman et al. (Kiciman and Wang, 2004) describe an algorithm to infer generalized correctness constraints on a system's configuration so that instead of diagnosing and resolving a particular problem that may arise, a problem can be characterized at a higher level as simply deviation from a correct configuration, and the solution is similarly to reconfigure the system to reach a known correct configuration. Aggarwal et al. (Aggarwal et al., 2004) suggests a way to reduce the risk of failure for the often error prone process of reconfiguration. It is based on data about the health of a system collected after certain operations are performed on it. They provide an algorithm that can analyze this data and suggest healthy sequences of operations that achieves a certain desired configuration.

2.4 Efficiency

Efficiency is the capability of a software product to "provide appropriate performance, relative to the amount of resources used, under stated conditions" (iso, 2001). This includes time behaviour which has to do with response and processing time and throughput.

Ruth et al. (Ruth et al., 2006) experimented with transparent autonomic migration and adaptation of virtual computational environments. A virtual computational environment, VIOLIN, is composed of virtual machines connected by virtual networks, which yields increased flexibility because of the high degree of decoupling it enjoys from the underlying physical infrastructure. They found that by combining migration of virtual machines between hosts with dynamic memory and CPU allocation to each an improvement in performance (execution time) of between 39 and 47 percent was achieved.

Zhang et al. (Zhang et al., 2007a) present an automated approach for locating performance problems by applying probabilistic inference to monitored service response times. They use a Bayesian network to model how end-to-end response times are linked to service elapsed (response) times. Their evaluation by simulation demonstrate significantly better accuracy than problem localization by random guessing. Their evaluation on a real world system shows that the cost of monitoring is negligible compared to performance gains available if the identified problems are solved, which may not always be possible.

Walsh et al. (Walsh et al., 2004) describe the use of utility functions to optimize resource allocation in a two-level architecture. A data center architecture consists of a set of application environments. Resources must be allocated globally among these environments, and locally within them. Allocations at both levels are performed by optimizing a utility function that attributes a value to a particular resource allocation. A resource arbiter manages the global allocation among application environments. Each application environment allocates resources locally within itself according to an application specific utility function. To accommodate change locally within the application environments or globally among them.

2.5 Usability

Usability is the capability of a system to be "understood, learned used and attractive to the user when used under specified conditions" (iso, 2001). This quality is different from the previously considered reliability and performance in that it is unclear it can be measured using a sensing system.

There has been some work on the human factors of autonomic systems that can inform the design of concrete autonomic applications (Lightstone, 2007; Barrett et al., 2004; Anderson et al., 2003). Lightstone (Lightstone, 2007) offers a set of concrete guidelines for self managing applications, including that they should never undo the explicit choice of an administrator, that they must always enable users to switch off autonomic behaviour, and that they should never force a user to make a choice simply because the developers could not. Barrett et al. (Barrett et al., 2004) performed ethnographic studies of systems administrators and emphasize the need for enabling awareness by operators, support them in rehearsal and planning activities and aid them in managing multitasking, interruptions and diversions. Anderson et al. (Anderson et al., 2003) goes further than Barrett et al. and explores the potential for autonomic computing to be made accountable, concluding that this is a matter of careful design.

If we return to the two main conceptual approaches mentioned in the beginning, the case for a biologically inspired approach to autonomous systems is alluring, however there are

also unresolved issues with respect to usability. Since no system is likely to be completely autonomous, a human operator must be able to step in and handle problems the system is unable to deal with itself. In such cases, it is crucial that the human operator is able to inspect the system and make sense of what is going on. When the system is based on explicitly represented plans and actions they are arguably easier to inspect than if they, in a sense, are not there. The importance of building systems so operators can make sense of them extends well beyond the handling of problems, and includes general awareness as pointed out by (Barrett et al., 2004). The concern for understandability is noted by Chen et al. (Chen et al., 2004) who uses decision trees as a representation around which to base their failure detection mechanism, based on machine learning, because they arguably are easier to make sense of than other options.

In their seminal paper on autonomic computing, Gane and Corbi (Ganek and Corbi, 2003) describe five degrees of autonomy in which increasing the degree of autonomy corresponds to requiring higher-level functionality in Kramer and Magee's reference model. Ganek and Corbi describes the division of labour between the autonomic features of the system and the human operator as divided along the high-level–low-level continuum. That means, in terms of the Kramer and Magee model, that a system administrator for many systems retain at least partial responsibility for higher level planning and execution tasks. Some of the tasks Barrett (Barrett et al., 2004) found are necessary to support can be characterized as high-level planning and execution tasks, however awareness may arguably extend to lower levels of the system as well.

2.6 Implications for Hydra

Our brief overview of detection and adaptation techniques has implications for further research in the area. Although many different techniques and architectures have been proposed more research is needed into how particular techniques relate and what is required to combine them. Most of the papers encountered in this survey are quite focused on a particular goal, purpose or technique in isolation. But a real-world system cannot be *just* reliable, efficient, secure or usable, but instead must meet a set of qualitative requirements in combination. A good understanding of how different techniques can be combined arguably becomes even more important when the set of qualitative requirements and the conditions for meeting them vary dynamically, as is the case for self-managing systems.

In Hydra we have addressed this challenge at the following points which we document in the remainder of this deliverable:

- In order to be able to combine techniques, the assumptions each technique makes must be explicit. Chapter 6 documents how the semantics of architectural elements can be modelled so as to enable that.
- A QoS ontology (along with some metrics) is proposed (section 6.3) which can make the internal qualities(e.g. performance) of a system subject to computational reasoning and therefore useful to self-management.
- We introduce an overall approach to self-management based on utility functions to enable high-level reasoning about self-management. This is described in chapter 8. Utility functions deal with the interactions among different techniques, and helps ensure that all possible dimensions of e.g. as number of possible reconfigurations is considered before one is chosen. By *considered* is meant that various aspects of sys-

tem QoS and state are modeled in our ontologies and are used as input to computing a utility function.

- Analytical work like the analysis in this chapter must be combined with experimental work on building real systems. However self-management in a pervasive system is difficult to experiment with. Therefore we have developed an architectural scripting language which can help set up complex configurations of a system, and describe the changes it undergoes at runtime. This is documented in chapter 5. It eases experimentation with systems that are complex to set up by automating the test part of iterative development. In additon, scripting an experiment makes it easier to document in a reproducible way.

TECHNIQUE	CONCEPTUAL APPROACH		ARCHITECTURE		QUALITY GOAL	
	idea	*mechanism*	*style*	*platform*	*quality*	*strategy*
GAIA[1]	plan-based	dyn. planning	layer	CORBA	reliability	STRIPS planner
Aura (Sousa et al., 2006; Garlan et al., 2002)	plan-based	dyn. plan	layer	task-based	reliab. & usab.	dyn. util. func.
Baresi et a[2]	plan-based	fixed plan	SOA	WS	reliability	dynamic (re-)binding
Kiciman et al. (Kiciman and Wang, 2004)	plan-based	fixed plan	OS-registry	Win XP	reliability	rec. correct config.
Aggarwal et al. (Aggarwal et al., 2004)	plan-based	dyn. plan	application env.	(not impl.)	reliability	healthy reconfig seq.
Goldsby et al. (Goldsby et al., 2007)	biological	evolution	none	AVIDA	generic	state machine as genome
Ruth et al. (Ruth et al., 2006)	plan-based	dyn. plan	virtual env.	cluster	efficiency	dyn. res. allocation
Zhang et al. (Zhang et al., 2007a)	plan-based	fixed plan	SOA	WS	efficiency	bottleneck loc.
Chen et al. (Chen et al., 2004)	-	diag. only	client/server	web server	reliability	m. learn: dec. trees
Srivastava et al. (Srivastava et al., 2004)	plan-based	dyn. plan	agent oriented	ABLE	reliability	PDDL planner
Walsh et al. (Walsh et al., 2004)	plan-based	fixed plan	virtual env.	cluster	reliability	util. func.

Table 2.1: Summary of properties for each technique. WS means web-services, SOA means service oriented architecture. The platform entry *task based* for Aura refers to a prototype specific environment in which tasks can be specified and realized using particular services/applications. The style *virtual env* given for Ruth et al. (Ruth et al., 2006) and Walsh et al. (Walsh et al., 2004) refers to an architecture in which virtual environments are provided in which applications can be executed. The virtualization of these execution environments allows for the dynamic allocation of resources between them. An entry *none* means that we found no assumptions to be made in this dimension. The term generic for the quality attribute of Goldsby et al's technique (Goldsby et al., 2007) is due to it being oblivious to the criteria which is used to introduce selectivity in the evolution. [1] (Chetan et al., 2005; Ranganathan et al., 2003; Roman et al., 2002). [2] (Baresi et al., 2007; Baresi and Guinea, 2007)

3 Self-* scenarios

We describe three cases where self-* functionality is useful. The cases are used in the remainder of this deliverable to exemplify the technical work that has been carried out. These scenarios extends the work of self-diagnosis scenarios (Ingstrup et al., 2008) to try to incorporate full scope of self-management.

3.1 Case 1: Repairing interface mismatches through reconfiguration

This scenario is from the Building automation domain covered by Hydra, which illustrates architecture based self-configuration (which can be considered as self-healing in this case) in that a problem with the components configuration is detected and repaired using the architectural concepts of services, interfaces and components.

Scenarios. Smith is a fan of new technologies and he has a smart personal assistant to automatically schedule sport/social appointments with friends based on his schedule and preferences. He has also a Smart Home system, which is equipped with self-managed heating and ventilating systems. While he is abroad for vacation, his home security system automatically turned on to the highest security level when detecting he is far away from home. While he is in a hotel, his home ventilating system detects an error through self-diagnosis. He is notified with an SMS on his mobile and he acknowledges that message to initiate a self-repairing process. A new service to resolve this error is found from a third party service provider, and begins to download some new software. Not surprisingly, the new service does not exactly match his system, and the problem is detected and resolved by invoking an adaptor service to seamlessly integrate the new service with the existing system.

Solution. To solve this problem, cases of interface-mismatch must be detected. When the interface mismatch has been detected, an attempt to repair it can be made.

3.2 Case 2: Reliability improvement through monitoring and rebinding of failing services

In the agricultural scenario (Ingstrup et al., 2008), the reliability of the control and monitoring system for the pig sty can directly affect the health of the pigs and therefore the profit of the farm. The following scenario is adapted from (Ingstrup et al., 2008).

Scenario. Bjarne is an agricultural worker at a large pig farm in Denmark. His daily routines include taking care of one of the slaughter pig stables, maintaining equipment, and helping with various jobs on the farm as needed.

To help him in his tasks, he carries around a PDA. The PDA has two primary functions. First, it allows him to record information about the tasks he carry out, such as medicating pigs and repairing machines, as well as making additional notes utilising context awareness

to record as much information as possible automatically. Secondly, it provides him with an interface to the farm monitoring system, where he can see and respond to alarms. The monitoring system is developed and based on the Hydra middleware. Many of the alarms are just notifications to show him that a problem has been fixed by the self-management subsystem of Hydra.

Solution. A major source of unreliability is failures. Whatever the cause of failures may be, their effect in a service oriented architecture such as found in Hydra based systems is often that one or more services become unavailable.

The contract between a service provider and its clients are captured in the interface description of that service. Therefore the clients of a failing service may remain operational during service failure if a replacement service with the same interface can be discovered and bound to those clients.

Therefore, enabling the system to detect the failure of services and subsequently re-bind the clients of failing services can help improve reliability which is an important quality attribute in the agricultural domain.

3.3 Case 3: A scenario for self-adaptation considering Quality of Service (QoS)

This scenario is used to show the adaption when there are multiple services utilizing different web service transportation protocols (currently SOAP over TCP or SOAP over UDP). We have tested that SOAP over UDP has better throughput over wireless network (e.g. Wifi network), and over the mix between wireless and wired network as shown in D5.9 (Sperandio et al., 2008).

Scenarios. Smith is now at his office and working using a notebook. He has some new songs from his favorite singer stored on his home PC. Now he wants to listen to these songs while working. Therefore he starts his Hydra based Smart Secretary to help him download the songs to his notebook. It is detected that both a Wifi connection and a Wired connection are available for the Smart Secretary, and download the songs within 10 seconds using the wired connection.

The other day, he is on a business trip in the airport. While waiting for the flight, he gets an emergency call from his colleague and is asked for uploading 50M bytes data to a server as soon as possible. The Hydra based application searched for Internet service, and two services are found. One is Wifi, 15$/hour, 1Mb/s, reliability of connection is 96%. The other choice is fiber cable Internet service bar, 50meters away, 60$/hour, 1000Mb/s, reliability is 100%. He is recommend option 2 in this situation as the data should be reliably upload to his company and as fast as possible.

Solution. To achieve the above scenario, QoS should be considered and the utility of a service should be considered in order to provide satisfying services, especially when there are multiple choices. For the above scenario, SWRL rules are used to help make choices based on the QoS requirements, the current network contexts in both Smith's office and his home. The Smart Secretary automatically switches to use SOAP over UDP based download service in this case based on the utility functions.

In the coming sections, we will demonstrate how to enable the above scenarios with the support from SeMaPS ontologies (including SWRL rules), and an architectural scripting language. An SWRL rule is developed to decide whether Smith is far away from home as a complex context. Another rule is used for detecting interface mismatch during the installation of a third party service for resolving this error, and an adaptor is added to the configuration to resolve the mismatch. And self-adaption rules based on QoS requirements are developed, and more self-diagnosis rules are developed.

4 The Hydra Self-* architecture

The Hydra architecture for self management follows the logical three-layer architecture (henceforth referred to as the 3L architecture) for self-managing systems proposed by Kramer and Magee (Kramer and Magee, 2007b) as adapted from Gat's three-layer architecture for robots (Gat, 1998b).

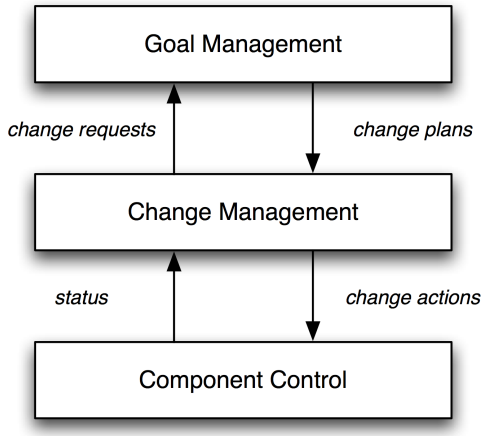

Figure 4.1: Three Layer Architecture Model for Self-Management

The 3L architecture is logical, and does not detail deployment, module or runtime views of a self-managing system. In addition to detailing these views, the Hydra architecture for self management achieves the following design objectives:

- The architecture should decouple the concerns individuated by the 3L architecture, and support runtime reconfiguration of itself, that is, the components responsible for goal management and execution should be dynamically pluggable.
- The existing functionality of Hydra managers must be leveraged to make the implementation lightweight and particularly targeted to Hydra systems.
- The previous survey shows that the current understanding how to combine techniques for optimizing particular quality attributes is poor, so the architecture should support an organized way to deal with system quality attributes and QoS.
- It must be easy to experiment with different self-management techniques, because the Hydra middleware during the project's lifetime should serve as a research platform.

4.1 Decouple syntactic and semantic layers of abstraction

From the three layers we can observe that there is a distinction between different layers of abstraction, or perspectives.

In the *component control* layer we view the system as consisting of services, components, devices and other concepts in the Hydra context. These entities can be manipulated

in a generic way, e.g. services and devices can be started and stopped, and components can be deployed to devices. The semantics of these operations are independent of *what* the services, components or devices do, their purpose in the particular application. Therefore the component control layer can be said to have its own self-contained semantics.

The *Change Management* layer is about managing changes, and to understand what this layer does we need only the generic concepts such as SWRL rules (in the case of Flamenco/SW). These are also the *only* concepts we rely on in the Hydra middleware's implementation of this layer. In a concrete application there will of course be application specific rules, but the notion of a rule itself is generic for all applications built with Flamenco/SW.

The *Goal Management* layer of the Hydra self-* middleware can likewise be understood and implemented in a generic way, using only generic concepts such as goals and plans. Again, a specific application will of course have domain specific instances of these concepts, specific goals such as "maintain 90 percent availability of service X".

To make the middleware generic, we need to maintain a separation of these generic semantics, those particular to Hydra software, from the application specific semantics of particular instances of services, plans, and goals. At the same time we must recognize that the manipulation of, e.g., the services in a particular application can be specific to those services if, for instance, two services cannot be restarted independently. The advantage of the 3L architecture is that application specific constraints on one level, e.g. services in the *Component Control* layer can be captured by instances at the level above. So if two services cannot be started in random sequence, that can be modeled using specific rules in the *Change Management* layer, or as application specific constraints of the plan generation in the *Goal Management* layer. This separation of different levels of abstraction is used in our implementation of each layer.

4.2 Overview of the Hydra Semantic web based Self-* Architecture and used tools/artifacts in different layers

The current implementation of the 3L architecture consists of a number of components, some of which are described in this deliverable, including the SeMaPS ontologies and self-management rules, Flamenco/SW for monitoring and reasoning, and the ASL scripting host which implements low-level manipulation of components. In addition to this, the web services generated with the Limbo compiler (Hansen et al., 2007) are able to generate service calling stubs in order to know the run time architecture of a underlying system and then conduct diagnosis for service fail. This is detailed in a published paper (Zhang and Hansen, 2008a).

Figure 4.2 shows the high level components used in Hydra self-* architecture to achieve self-management, where the dependencies of components will be detailed in Chapter 9. The interaction of different layers are through eventing via the Event Manager, therefore we have eventing clients for the different layers.

For the Component Control layer, we are using Limbo to generate service message calling stub code in order to report run time service calling (ServiceMessage component, in corresponding to MessageProbe ontology introduced in Chapter 6), and also generate state machine code (StateMachine component, in corresponding to StateMachine ontology introduced in Chapter 6) in order to conduct state based diagnosis at run time. Also resources are going to be monitored at run time, and OSGi components (as the Hydra component model) are monitored to check their architectural constraints. To enable the action of component loading and unloading, and managing the test bed, we have an architectural script

language and a corresponding ASLhost bundle that achieves this.

The Change management layer will be responsible for listening to events reported from the Component Control layer e.g. device state changes, and service calling, which will then trigger the execution of self-management rules. Therefore in this layer, the components are relying on the SeMaPS ontologies and SWRL rules.

The Goal management layer adopts the utility function based planning detailed in Chapter 8. Planning is a process for multi-objective selection of best candidates of services and plans, for example to achieve lowest cost, and at the same time better throughput with good performance, in a global manner. We are going to use a Genetic algorithm as the underlying planning algorithm in the Goal management layer.

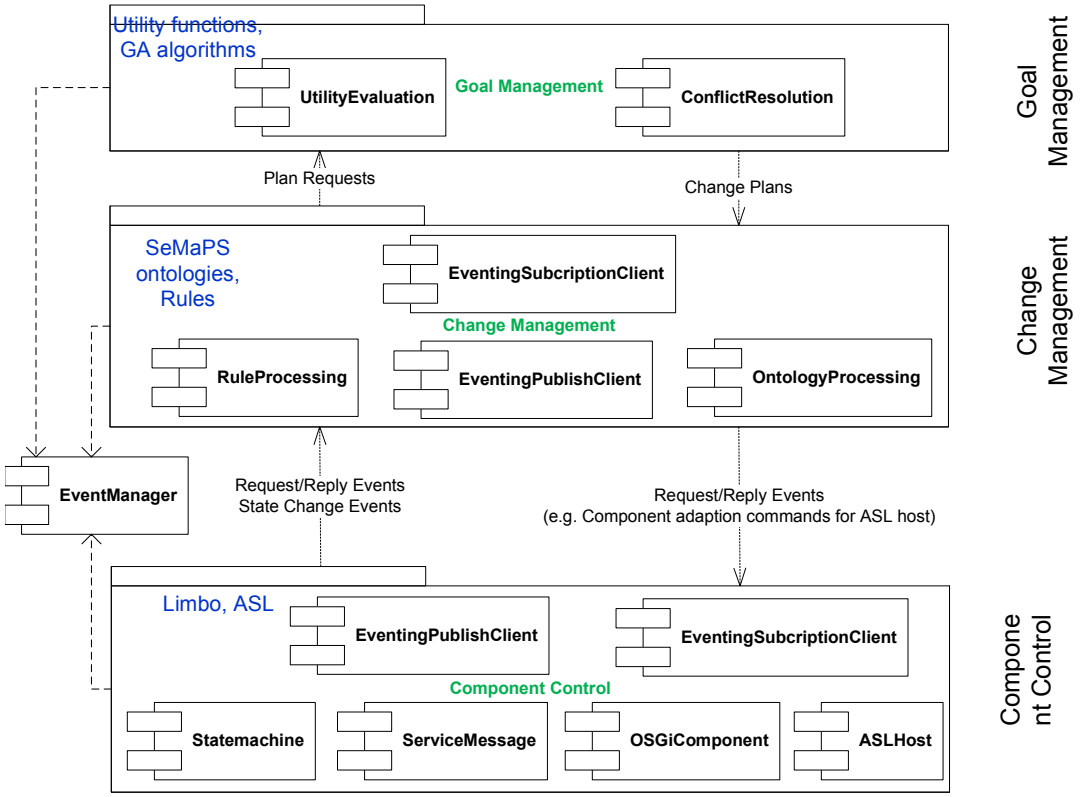

Figure 4.2: An overview the overall architecture and components for achieving self-management goals

4.3 The self-* architecture in relation to the Hydra architecture

In this section, we will clarify the relationship of the self-* architecture with respect to the Hydra software architecture, based on its current situation of both of them. Ideally every manager should have this architecture to be a kind of self-management manager itself, and self-manage component can act as a supervising manager to coordinate the high level

actions. Currently, the self-* architecture extends the hydra architecture as documented in deliverable D3.9 (Eisenhauer et al., 2008) in a number of respects. Here we describe them according to which layer they stem from.

The component control layer:

- poses a requirement to services by requiring them to be able to report their state and service calling messages; This is automatically achieved when using the Limbo compiler to generate web services for event publishing via the Event Manager, including the generation of state machine code (with Limbo state machine plugin), and service message probes code (with Limbo Probe plugin).
- the ASL interpreter+actuator does not as such pose requirements to the architecture, but to the component model and platform it is implemented on. Not all component models support the manipulation of programs in terms of components, services etc. Some may not allow dynamic rebinding.
- In an ideal situation, the context manager should monitor the underlying context changes, and report to every Hydra manager of the current usage contexts, and then this Hydra manager will itself adapt to the current situation by taking consideration of the current QoS requirements from the QoS manager, which monitors the QoS attributes. Put the Network manager as an example, the context manager detects that battery is low, Network manager should disable some of its feature, or switch to JXME[1] version of it (not available currently and is under investigation by TID). In this respect, the Hydra architecture should be improved in the future.

Change Manage layer:

- in order to reason about self-adaption, self-configuration and other self management activities, it is necessary that services and components explicitly declare their dependencies, and also its specific interface dependency details, like interface signatures. We could then make use of the set of architecture ontologies to reason whether the current configuration is correct or not.
- as said, it is advantageous that every manager has its own self-management architecture including the change management layer as explained in the last item of the Component control layer.

Goal management layer:

- As the goal management layer is supposed to be computationally heavy, therefore it is reasonable that only the self-management component (Flamenco) has this layer, while other managers have only the basic Component control and Change management layers. In this case, the current Context manager needs also to be extended to cover this, to fully support the contexts outlined in Chapter 6, and delivery these contexts properly.
- One of the most important features of the self-management extensions to the architecture is that self-* features can modify the quality attributes of a system when a given goal is chosen. Therefore this may require that QoS manager need to monitor the changes of QoS attributes and report that accordingly.

[1] https://jxta-jxme.dev.java.net/

4.4 Connections with other Hydra components

Self-management components need to collaborate with other Hydra components. Take the case of scenario number 1 given in chapter 3, Figure 4.3 shows how the layers in the architecture should interact (in a high level view). The Hydra Discovery manager need to discover the needed service, and there found components needs to be bound/unbound, the ASL bundle would do executed that as specified in a script. The interactions of the self-management components themselves will be elaborated in Chapter 9.

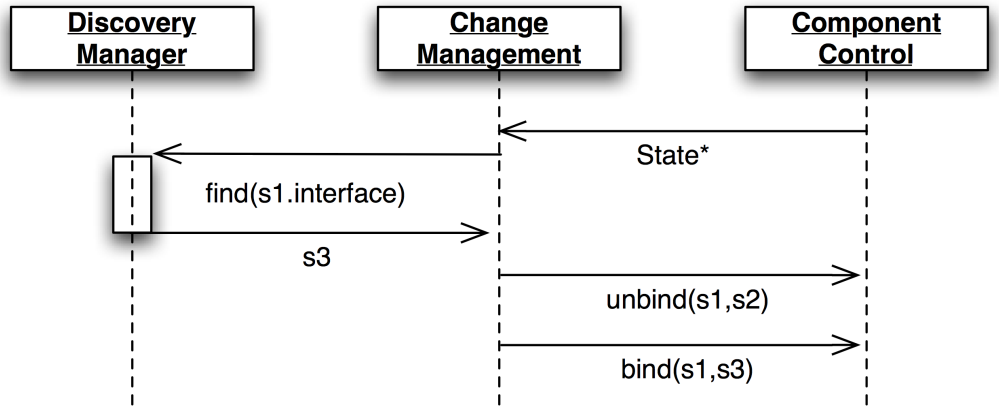

Figure 4.3: The interactions of the component control layer and the change management layer for scenario 2.

Hydra is proposing the semantic-web based self-management approach. For the availability of OWL/SWRL APIs reasons, the self-management is using Protege-OWL/SWRL APIs, and Ontology manager is using Jena. Ideally the ontology manager should act as the single point for ontologies, therefore we need to work on that Ontology manager covers the full set of SeMaPS ontologies.

Context manager should be the source of the triggering of the self-management, as self-management highly depends on contexts (especially the run time contexts, device resources, user preferences etc.). But currently it could not achieve this as none of these contexts are available in the Context manager, and we heavily rely on the Hydra Event manager as shown in Figure 4.2 for the achieving of self-management.

QoS manager has the close relationship with self-management, which should report to the self-management component of the monitored QoS as required. The planning layer need to get what are the required QoS parameters for certain situation then, planning for the selection of services (for self-adaptation, self-configuration) (e.g, by evaluating the utility), for service composition evaluation by evaluation different plans for combinations of services, for deploying services on node based on different resources, for self-optimization plans, all with the QoS considerations. This will be detailed when the design and implementation of the QoS manager are ready.

5 Architectural script language design, implementation and its applications

The past decade of research in software architecture description has been prolific (Medvidovic and Taylor, 2000), but only in so far as the modeling of static aspects of software architecture is concerned. While this work has proved fruitful, a number of recent trends suggest a growing need for the ability to model dynamic aspects of architectures. We suggest *architectural scripting* as a way to express changes to an architecture.

An architectural script consists of sequences of operations on a runtime architecture, such as deploying a binary component to a device or instantiating a service from a binary component. We term this set of operations the architectural scripting language, or ASL.

Architectural scripting serves two purposes in Hydra and helps achieve the design objectives set in section 4:

- It is a useful way to describe and reason about the reconfigurations that a self-managing system may perform. Thus an interpreter+actuator for architectural scripting provides part of the implementation of the component control layer in the 3L architecture adopted for Hydra self-*.
- To ease both the initial setup and ongoing management of concrete test-scenarios/configurations in our testbed for autonomic computing. Thus it makes it easier to experiment with self-* systems by automating part of the configuration setup required to actually run a distributed prototype.

Architectural change has received little attention, and therefore it is worthwhile to be clear about how we have chosen to conceptualize it. In section 5.1 we formalize our notion of architectural change as consisting of a series of discrete steps, and highlight the features of our conceptualization.

Next, we will elaborate two applications of architectural scripting within Hydra: a testbed for autonomic computing and an implementation in the Component Control layer. In section 5.2 we describe implementation of ASL operations that have been made in ANT as a first step towards making a testbed for autonomic computing. In section 5.3 we describe how architectural scripting forms part of the implementation of the component control layer, implemented as a service in the equinox OSGi platform.

5.1 Modeling architectural change

We model changes to an architecture as consisting of a series of discrete atomic steps which we term operations. Operations can be combined to achieve particular changes to an architecture. Table 5.1 shows a catalog of the operations that makes up ASL.

Operations and Architectural entities. An architecture is represented as set of typed entities where the set of types is $T = \{Component, Connector, Service, Interface, Device\}$. The parameters of an operation are *handles*, named variables which can be resolved in an implementation dependent way to identify precisely one entity in the architecture.

The namespace of parameters is organized in two hierarchical scopes, with devices constituting the top level and components the subordinate level within which services are uniquely identifiable. Hence, to resolve (find the unique identity) a service-handle which

operation name (parameters)
deploy_component (component, device)
undeploy_component (component, device)
start_service (service, component,device)
stop_service (service, component, device)
start_device (device)
stop_device (device)
bind (service, interface, connector, interface)
unbind (service, interface)
print_status (device)

Table 5.1: The operations supported in the ANT extension for architectural scripting. The operation print_status does not have any effect on the architecture so it is only provided for the convenience of the programmer.

should be started, it is necessary that the *start_service()* operation takes as parameters the component the service is found in, and which device that component is deployed to.

Atomic execution. Operations come from the set shown in table 5.1. It is important to note that all the operations which have an effect on the architecture (that is, all of them except *print_status*) come in pairs such as *start_service* and *stop_service* where the effect[1] of one is reversible by the effect of the other. That is significant because it enables roll-back of a partially executed script, as required for atomic execution of a distributed script implemented with two-phase commit.[2]

Semantically correct operation sequencing. Semantic dependencies can exist among operations. An operation that starts a service will logically be dependent on whether that service has previously been deployed. According to the principle of separating layers of abstraction (in this case syntactic and semantic) ASL does not explicitly include any facilities for ensuring execution in accordance with semantics of components, except that the operations are synchronous and executed in non-overlapping sequence. When applied in self-management, ASL will implement the component-control layer only, and rely on the semantic tools for ensuring that only correct scripts are submitted to it. When applied for testing and configuration setup, the dependencies among groups of operations can be enforced by correct distribution of operations among a set of interdependent ANT-tasks.

Abstract format of an architectural script. The format of an architectural script is shown below using an abstract syntax. The syntax is abstract because it may exist in one of two concrete forms:

1. When used for test/configuration management, a script is written in the ANT XML format using the ANT-bindings provided by the testbed.
2. When used for self-management, a script is an instance of the java class ASLScript send to the ArchScriptHost service.

[1] if any: deploying a component that is already deployed has no effect, so it should not be attempted undone by undeploying the component

[2] Atomic script-execution has not been implemented in the present prototype.

The syntax of an architectural script follows the following grammar:

```
<script>    := <operation>*
<operation> := <opname><parlist>
<opname>    := "init_device",
               "start_device",
               "stop_device",
               "init_component",
               "deploy_component",
               "undeploy_component",
               "init_service",
               "start_service",
               "stop_service",
               "print_status",
               "bind_services",
               "unbind_services"
<parlist>   := <parameter>+
<parameter> := String
```

In order for a script to be semantically correct, a handle to a component, service or device must be initialized before it is used. Currently we do not support control-structures like a conventional programming language. It is possible to use control-structures in ANT. Experience applying the language will guide future extensions.

5.2 Architectural scripting for test/configuration setup

We apply architectural scripting to manage the setup of particular configurations of distributed systems, and we have done so by extending the well-known tool ANT with the ASL operations. This is useful because:

- Tools like ANT and Make are convenient, but only cover the development-time—or the module view in architectural terms; they are not specifically geared towards helping set up and run complex deployments of a system.
- For experimental researchers in software engineering, using a scripted routine to run a program can help ensure reproducibility of experiments. That may not be an issue for standalone applications, but it is more and more often the case that experiments concern several interacting components/services, and are distributed. This is specifically the case in Hydra.
- For testing, an architectural script provides an operational way to describe test scenarios, e.g. a failing device can be modeled by stopping a device and starting it again.

In building the prototype we have pursued a design which can evolve into a testbed for distributed self-* systems. This is briefly described below.

5.2.1 Towards a testbed for distributed/self-* systems

When testing software in a distributed system, it is cumbersome to set up the test environment. For autonomic computing this problem is aggravated because an additional feature that is important to test for autonomic systems is their capability to deal with changing configurations (due to e.g. failing devices). Architectural scripting is part of the solution to this because it provides an operational way to describe the behaviour of a testbed by defining, for instance, when a simulated device fails.

ASL ontology	*OSGi platform*
device	JVM running equinox
component	bundle jar-file
service	running bundle
interface	service-interface
connector	a set of running bundles

Table 5.2: Mapping the abstract ontology of the testbed to the OSGi platform.

A testbed for autonomic computing can be easier to experiment with than a real system in several ways. Firstly, it can abstract away some details that would be required in a real prototype, and which are tedious or otherwise expensive to deal with, but which may not be relevant to the goal at hand. This may be setup of particular devices whose software interface is instead provided by a simulated or virtual device. Secondly, it can provide a library of frequently used building blocks, such as routing algorithms, adaptation strategies or similar. Thirdly, it may allow greater control of experimental parameters, thus making it easier to perform experiments in a way that is reproducible.

In the current prototype focus has been on implementing the ASL operations in ANT, for the Equinox OSGi platform. However the design is made in such a way that it needs minimal modification in order to support new platforms/types of devices, because that is a priority in the current stage of Hydra. The prototype only relies on Equinox specific functionality for starting the platform; the implementation of all other operations than *start_device* rely only on the OSGi standard.

Concretely, the ANT-tasks are implemented in terms of an abstract architectural ontology shown in figure 5.1. This ontology is in fact can be modeled by importing the Device ontology, and the set of architecture ontologies detailed in Section 6.4. These concepts are mapped to specific platforms when the ASL operations are executed on specific architectural entities. For OSGi, the mapping is as shown in table 5.2 according to the OSGi specification. The *start_service* operation supports passing key-value lists of string based parameters. A virtual device is a programmatic object in the memory of a physical device.

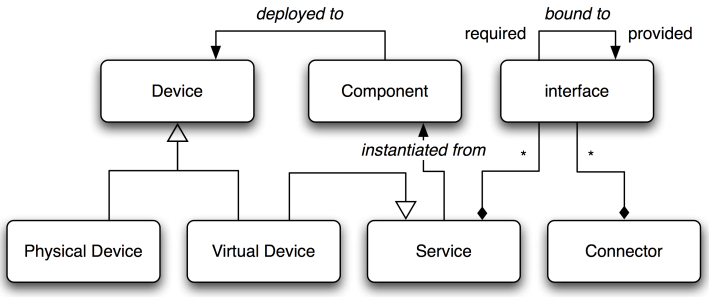

Figure 5.1: The abstract ontology guiding the testbed's logical structure.

For testing, a device may be purely virtual, such as a simulator, or virtual machine. The only device currently supported is the Equinox OSGi platform running on the JVM; it is an example of a virtual device. A component is a binary unit of distribution. A service is a unit of runtime software accessible by other services. A Service provides one or more interfaces.

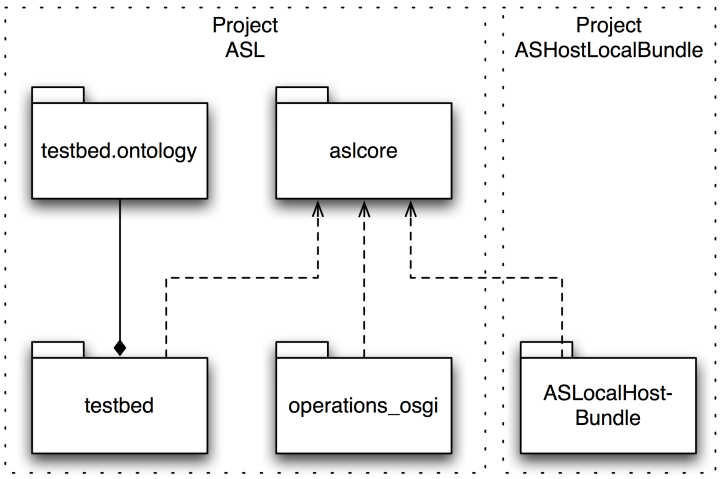

Figure 5.2: A module view in UML. It shows the organization of the ASL code base. The boxes drawn with dotted lines denote how the packages are distributed onto eclipse projects/SVN modules

5.2.2 Testbed implementation with ASL

In the prototype each operation is implemented as a task in ANT, following the standard procedure described in the ANT manual [3]. This way, invocations of operations can be grouped together according to what goal they achieve. In architectural scripting, we can make a target in such a way that it ensures the satisfaction of a constraint on an architecture. For instance, a target could be to ensure a particular service is running. That target, in turn, might depend on the deployment of the component from which that service is instantiated. Deploying that component would in turn depend on building the source-code, and so on. In this way our notion of architectural scripting may be seen as an extending the application of ANT to cover the whole life cycle of architectural elements, including programming, deployment, and runtime.

The classes providing an ANT task for each ASL operation are programmed against the abstract ontology, and a parameter is used to select the implementations particular to a given platform.

Module view. Figure 5.2 is a module view showing the organization of the ASL codebase. The ASL codebase consists of the following packages:

testbed.ontology This package defines as interfaces the basic concepts used in the ontology of ASL; it includes component, device, service. The set of operations making up ASL are mapped to methods on these interfaces; most of them, such as deploy_component are methods on the Device class.

aslcore defines the classes representing a script instance, the operations, the interpreter of a script and the interpretation context which is responsible for registering how particular (e.g. component-) identifiers are bound to entities in the actual, running system.

[3] http://ant.apache.org/manual/index.html

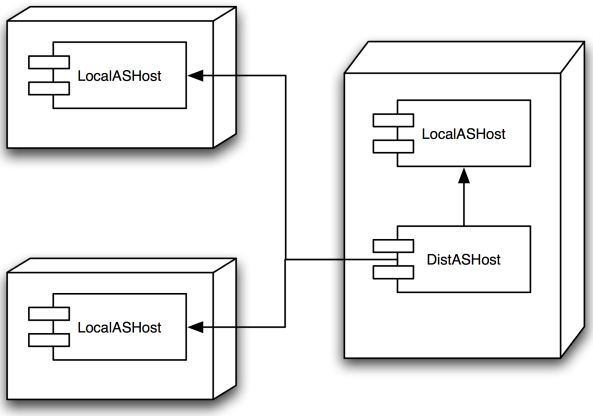

Figure 5.3: An example deployment of a distributed scripting host. Each device has a local engine which executes a script on the entities deployed to that device. The distributed scripting host will interpret a script which operates on a set of entities distributed across a range of devices, in this case three.

testbed defines the classes making up the testbed; currently it only supports the Equinox OSGi platform. It provides device specific implementations of the Service, Component and Device interfaces found in the testbed.ontology package.

operations_osgi defines the classes necessary for introducing the ASL operations as ANT tasks.

Runtime and deployment view. When used as a test/configuration management tool, ASL is used through the ant bindings. These are distributed in terms of a jar-file providing the classes that plugs into ANT, and an ANTscript which sets up the ASL tasks and defines them for use in the targets of a build file.

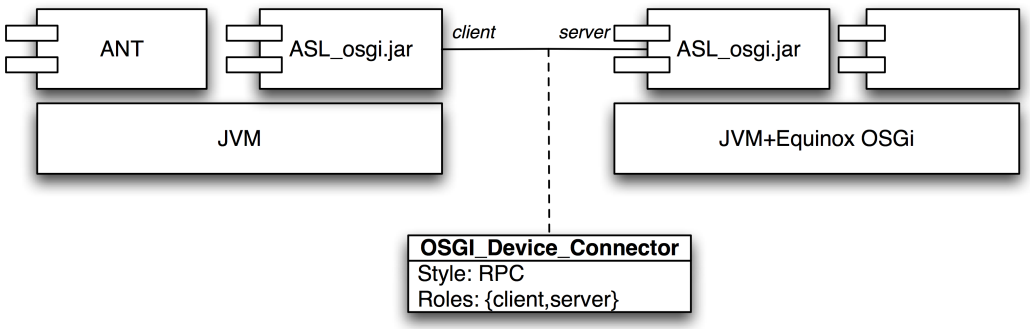

Figure 5.4: A runtime view showing what configuration may result when an ant script is executed to start a new device. The new device is an instance of the Equinox OSGi platform. The ANT tasks communicates with the platform using a custom RPC connector.

Tool support for ANT based architectural scripting As applications written in osgi often depends on a host of other bundles, the number of bundles that has to be installed can grow quickly. Given the verbose format of the ANT XML format in this case, we provide a simple tool to generate ASL scripts. Given a directory for a set of jar-files that are osgi bundles, it will generate the *init_component*, *deploy_component* and *init_service* operations, that is, the basic setup for installing the bundles and starting them.

We label the generator 'relaxed' because it does not guarantee that the generated series of operations will work in the actual system. Making such guarantees is possible if we represent all dependencies among components, services, devices etc. However in practise most third-party bundles does not obey that requirement. Such an assumption is unrealistic for most real-world systems and, moreover, because the tool aims to assist rather than take over the task of writing an architectural script the tool is useful even when the conditions for correct script generation are not met. In such cases the tool can produce the bulk of he script code, which can then be tweaked manually so that it sets up the initial configuration in a way that respects all actual dependencies, whether they are explicit or not.

Please refer to the tutorial on architectural scripting found in Deliverable 12.5 (Fernandes et al., 2008) for details of how to use the ASL extensions to ANT.

5.3 Architectural scripting for self management

To apply architectural scripting in self management, an API must provide an implementation of the ASL operations. In the current prototype, the interface through which the ASL runtime is accessed is a service with the following interface:

```
public boolean executeScript(String script);
public boolean executeScript(ASLScript script);
```

The second method takes a parameter which is a parsed script; it is functionally equivalent to the first but has better performance when the same script is executed repeatedly because it allows parsing to be done one time only.

In the previous Section 9.1.1, an example was given which showed how the runtime elements in the overall architecture interact to realize the first of our scenarios. Continuing that example, the following is a script as it might be used in the scenario. The services s1, s2, s3 are the client, the failing provider and the valid replacement service respectively.

```
init_service(s1,"eu.hydra.hellouser");
init_service(s2,"eu.hydra.helloprovider1");
init_service(s3,"eu.hydra.helloprovider2");
unbind_services(s1,s2,"eu.hydra.asl.interfaces.Hello");
bind_services(s1,s3,"eu.hydra.asl.interfaces.Hello");
```

Initially, the three service-handles are defined. The first argument names the handle, the second names the implementation-specific string used to identify the service. For the Equinox OSGi platform, that is the symbolic name of the bundle.

In this case, no device is specified as scope for the services, so the local host is assumed. In general, distribution of a system among several nodes has implications for how to execute reconfigurations. A script is assumed to apply to one device only, so that the scripting host service only executes operations whose execution (though not necessarily effects) are local to the device it is deployed to. The reason why that assumption is made is twofold: First, it simplifies the scripting host service significantly so that it can remain light-weight and still be used on small devices. Second, it does not incur an inherent cost of functionality, since an interpreter for a distributed script can be built on top of a set of local scripting hosts,

as shown in figure 5.3. In particular, scripts written in the ANT extension are distributed according to this scheme.

This section documents the structure of the current implementation of ASL, in terms of the module view of the source code, a deployment view which shows how ASL is set up for use with either ANT or in for self-management. Finally, a runtime view shows the runtime components and connectors.

Runtime view. used for self-management, the ASLocalHostBundle must be used. A runtime view of that is shown here:

Figure 5.5: A runtime view showing the dependencies of the ASHostLocalBundle to other bundles. The dependencies to Axis and the EventManager are optional because the bundle can in principle be use as a pure OSGi service. The Axis bundle enables the ASL host bundle to make the interface available as a web service.

6 Ontologies for self-management

Self-management is intrinsically complex. It is important to know the underlying contexts for triggering self-management activities, and then take the quality of service into consideration for these activities. Therefore, the underlying contexts, the knowledge for the contexts including QoS, are very important. The application of semantic web ontologies is arguably the most promising approach for context modeling and reasoning when compared to its counter parts, and is therefore chosen as our context modeling approach in relation to self-management (Zhang and Hansen, 2008a) .

To achieve self-management, the dynamism should be considered as a pervasive system is dynamic in nature. This dynamism should be considered in context ontologies. Also from the architecture point of view, a software system should remain stable and consistent after the self-management actions. That is to say, they should follow necessary architectural constraints. All this knowledge is considered in our self-management ontologies, which we call Self-management for Pervasive Service (SeMaPS) ontologies.

6.1 Why adopt OWL-DL and SWRL

OWL and SWRL adopt the Open World Assumption (OWA). It asserts that knowledge of a system is incomplete, which means that if a statement cannot be inferred from what is expressed in the system, it still cannot be inferred to be false. In the OWA, statements about knowledge that are not included in or inferred from the knowledge explicitly recorded in the system may be considered unknown, rather than wrong or false in the closed world assumption. The OWA applies to the knowledge representation where the system can never be known to have been completely described in advance, which is quite consistent with the characteristics of pervasive computing systems, as pervasive systems are intrinsically open and dynamic. Therefore it is natural to apply the open world assumption to the pervasive computing system where the characteristics of OWA can be utilized to build the concept of open world software (Baresi et al., 2006). We envision that semantic web based self-management is advantageous for achieving self-management goals (Zhang and Hansen, 2008a).

6.1.1 Justifying OWL-DL and SWRL strengths

Table 6.1 shows the comparisons of various context modeling languages, including XML, RDF(S), OWL, and SWRL. As we can see from the table, OWL has the most powerful capabilities for modeling, and SWRL can additionally express constraints which are not easily expressed in OWL. We have chosen OWL-DL as the basic for ambient intelligence because:

- The key to self-management is knowledge of the underlying systems, OWL-DL is powerful for knowledge modeling for both dynamic knowledge and static knowledge
- Open and extensible as needed, and also inter-operable in the Internet scale as required for real life pervasive systems
- Formal description logic and inference is decidable, able to provide reasoning capabilities in open world settings with usable and acceptable performance

- (De facto) standard, well-adopted by pervasive computing projects (for example Amigo[1], MUSIC[2]) for achieving context-awareness
- Potentials to make use of semantic web service (OWL-S) to automatically invoke and compose web services

Table 6.2 shows the rule comparisons currently available in terms of standardization, tool support and relationship with OWL. As we can see, there are many choices for rule languages. But when taking account of their maturity and tool support, and its adoption among the reasoners, it is obvious SWRL should be chosen because:

- It is expressive enough to allow both global and local constraints span across multiple concepts and concept properties (relationships)
- It is is decidable using DL-Safe rules (rules written only on known instances in ontologies)
- SWRL is OWL specific, and is basically a combination of OWL and RuleML
- It is De facto rule standard for semantic web rule specification
- It is extensible with new built-ins, and rules can be parameterized and changed dynamically at run time, and executed as rule groups to resolve rule conflicts and improve performance (shown in our evaluations)

Table 6.1: Feature comparisons of different context modeling languages

	XML DTD	XML Schema	RDF(S)	OWL	SWRL
Bounded lists				X	
Cardinality constraints	X	X		X	
Class expressions (unionOf, complementOf)				X	
Data types		X		X	
Enumerations	X	X		X	
Equivalence (properties, classes, instances)				X	
Formal semantics (model-theoretic & axiomatic)				X	
Inheritance			X	X	
Inference (transitivity, inverse)				X	
Qualified constraints				X	
Reification			X	X	
Global Constraints across multiple properties and instances					X
mix classes and properties directly					X

Table 6.2: Feature comparisons of different rule languages

	SWRL	WSMLRule	WRL	SWSLRules	ERDF rules	RuleML	R2ML
Relationship with OWL	OWL extension	NA	translate to a subset of OWL	NA	RDF	NA	NA
Standardization	W3C submission	NA	W3C submission	W3C submission	NA	NA	NA
Tool support	Pellet[3], RacerPro[4] ...	IRIS[5], MINS[6]	NA	NA	Jena[7]	DR-Device[8]	NA

[1] http://www.hitech-projects.com/euprojects/amigo/
[2] http://www.ist-music.eu/

6.1.2 Weaknesses of OWL-DL and SWRL

The disadvantage of OWL-DL and SWRL is that operations (such as queries) are computationally complex which may cause a performance bottleneck. Therefore it is important to measure performance to check whether it is suitable for the usage to achieve self-management goals. Another disadvantage is that SWRL is limited to unary or binary relationships, rules may become verbose for human readers, although SWRL is primarily intended for machine processing.

SWRL (and OWL) support monotonic inference only, which does not support negation as failure as it would lead to non-monotonicity. SWRL does not support disjunction either. But it is not hard to work around this. For example, RacerPro native rule language offers NAF negation and classical negation as well. "True" disjunction is offered for concept/class query atoms and role/object property query atoms, and "union" (of sub-query results) for arbitrary queries.

6.1.3 Basic introduction to OWL-DL and SWRL

To facilitate the reading of this deliverable, we will briefly introduce SWRL, and OWL-DL syntax, though SWRL is already introduced in former deliverable e.g. D4.3 (Ingstrup et al., 2008).

OWL-DL is the description logic $SHOIN$ with support of data values, data types and datatype properties, i.e., $SHOIN(D)$. The DL syntax and its corresponding OWL meaning is shown in Table 6.3, and the details of DL can be found in (Nardi and Brachman, 2003).

$\top$	Super class of all OWL classes
$N1 \sqsubseteq N2$	N1 is sub-class or sub-property of N2
$C1 \sqsubseteq \neg C2$	Class C1 and C2 are disjoint
$C1 \equiv C2$	Class C1 and C2 are equivalent
$C1 \sqcup / \sqcap C2$	Union/intersection of Class C1 and C2
$\top \sqsubseteq \forall P.C$	Range of property P is C
$\forall / \exists P.C$	allValuesFrom/someValuesFrom restriction
$= / \leqslant / \geqslant nP.C$	cardinality/minCardinality/maxCardinality

Table 6.3: Summary of OWL syntax

SWRL is a W3C recommendation for the rule language of the Semantic Web, which can be used to write rules to reason about OWL individuals and to infer new knowledge about those individuals. A SWRL rule is composed of an antecedent part (body), and a consequent part (head). Both the body and head consist of positive conjunctions of atoms. A SWRL rule means that if all the atoms in the antecedent (body) are true, then the consequent (head) must also be true. SWRL is built on OWL DL and shares its formal semantics. In our work, all variables in SWRL rules bind only to known individuals in an ontology in order to develop DL-Safe rules to make them decidable. SWRL provides built-ins such as math, string, and comparisons that can be used to specify extra contexts, which are not possible or very hard to achieve by OWL itself. In a SWRL rule, the symbol "$\wedge$" means conjunction, "$?x$" stands for a variable, "$\rightarrow$" means implication, and if there is no "?" in the variable, then it refers to a specific instance.

6.2 SeMaPS ontology structure

Some ontologies (Device, HardwarePlatform, Malfunction, StateMachine) are discussed in D4.2 (Hansen et al., 2007) and D4.3 (Ingstrup et al., 2008). We briefly mention them here

to make the description complete. A first draft of the QoS ontology is briefly discussed in D4.5 (Scholten and Shi, 2008). We present a more comprehensive version here, but may improve it in future as the project progresses.

The SeMaPS ontologies have some unique features compared to the existing pervasive computing ontologies, these include:

- Incorporation of user-centered concepts, such as user hobby and habit information, user preferences.
- Incorporation of runtime and dynamic context. A StateMachine ontology is used to model device runtime status and then is used to conduct state-based diagnosis. A MessageProbe ontology is used to report service calling relationships in order to know network conditions and device aliveness. Other ontologies are also involved in the modeling of dynamic context.
- Complex contexts that span multiple instances and properties, which are not expressible by OWL-DL ontology itself are described by SWRL, which adds more expressive capabilities to OWL and is extensible through built-ins.
- Semantic modeling of dependability of middleware, especially the self-recovery related concepts and properties in ontologies, which support decisions in the self-diagnosis process using theoretical reasoning.

The relationships between the SeMaPS ontologies are illustrated in Fig. 6.1.

The Location ontology models where events happen, in which Set-based, Semantic-based, and Graph-based modeling of location are encoded in a separate ontology based on (Flury et al., 2004). The Time ontology models time and date related knowledge, for example the time zone concept, which is developed based on OWL-Time[9].

The Person ontology encodes the majority of the user-centered concepts. It contains information about a person's hobbies like sport (with *badminton*, *tableTennis* and *soccer* as instances), color (e.g *skyblue*), music (various styles such as *CountryMusic* and *RocknRoll*) and also mood (*happy*, *normal*) descriptions. Depending on a user's mood, he/she may want to listen to different kinds of music, for example listening to country music if the user is in a happy or in normal mood.

The Device ontology is used to model equipments in pervasive computing, including SoftwarePlatform and HardwarePlatform. It presents *Device* (as a concept) type classification (e.g. mobile phone, PDA, Thermometer), which is based mainly on the device classification in Amigo project ontologies (IST Amigo Project, 2006). To facilitate self-diagnosis, there is a concept called *System* to model a system composed of devices to provide services. A corresponding object property *hasDevice* is added, which has the domain of *System* and range as *Device*. The *Device* concept has a data-type property *currentMalfunction* which is used to store the inferred device malfunction diagnosis information at run time.

The HardwarePlatform ontology defines concepts such as *CPU*, *Memory*, and so on, and also relationships between the devices (in the Device ontology), for example *hasCPU*. This ontology is based on the hardware description part from the W3C deliveryContext ontology[10]. Power consumption concepts and properties for different wireless network are added to the HardwarePlatform ontology to facilitate power-awareness, including a *batteryLevel* property for monitoring battery consumption at runtime.

[9] http://www.w3.org/TR/owl-time/

[10] Delivery Context Overview for Device Independence. http://www.w3.org/TR/di-dco/

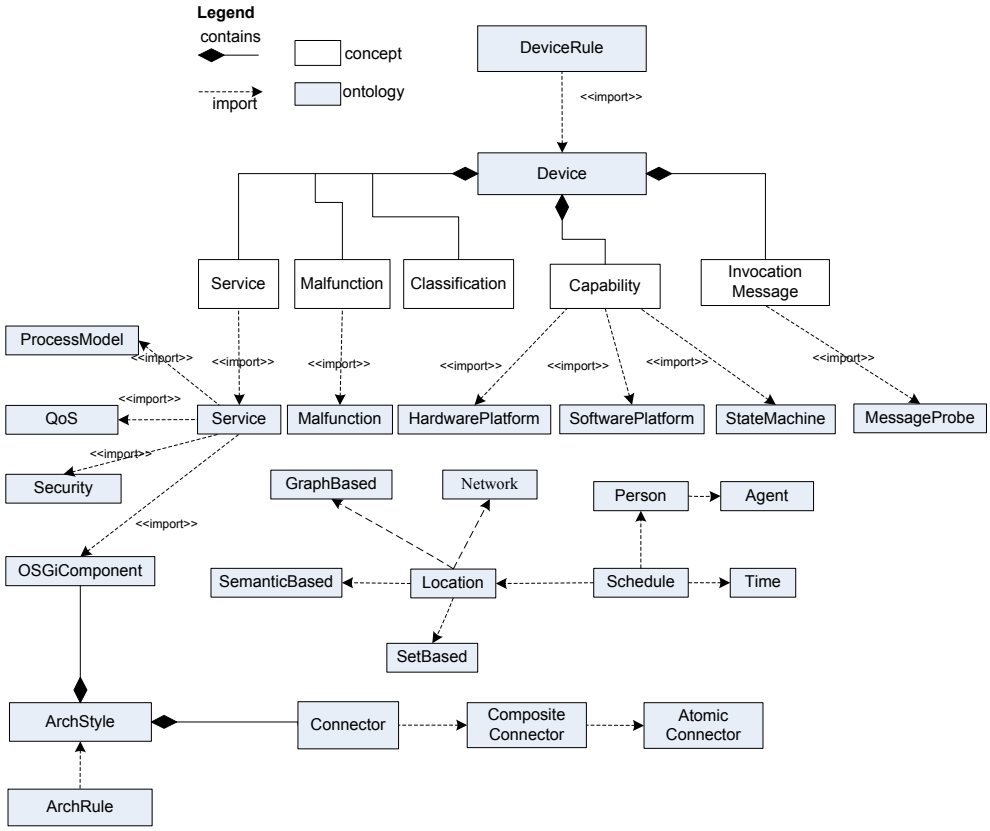

Figure 6.1: Structure of SeMaPS ontologies

The device Malfunction ontology is used to model knowledge of malfunctions and recovery resolutions, it also provides classification of device malfunctions (for example, *BatteryError*). The malfunctions are classified into two categories: *Error* (including complete device failure) and *Warning* (including function scale-down, and plain warning), according to its severeness. There are also two other concepts, *Cause* and *Remedy*, which are used to describe the origin of a malfunction and its resolution.

Three different classes of schedule are modeled in the current Schedule ontology, namely *Appointment*, *Meeting* and *Course*, which have object properties *participant* and *holdLocation*, and datatype properties *starts* and *ends*, denoting the time for the scheduled event to start and end, respectively. This information is used to schedule appointments, for example a sport appointment.

A more detailed but simplified view of the SeMaPS ontologies is depicted in Figure 6.2.

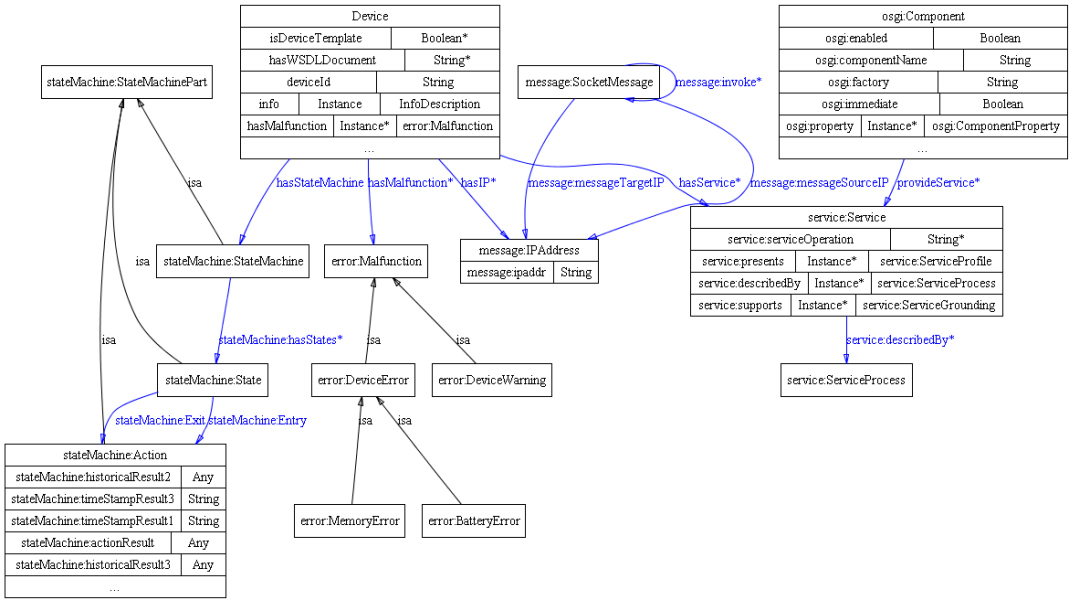

Figure 6.2: Partial details of the SeMaPS ontologies facilitating self-management

6.3 QoS ontology

QoS refers to a level of a service that satisfactory to some user, which is not necessarily the best level of service, but the one meeting the user requirements, which is then described by Service Level Agreement (SLA). For Hydra, as a pervasive middleware, QoS should cover some fundamental elements, including transportation network characteristics, power and energy consumption, and underlying service properties. At present stage for this deliverable as per the requirements for self-management, we list the following elements as the necessary QoS parameters that should be considered:

- Bandwidth, or throughput
- Latency

- ErrorRate
- Availability, including both network availability, and service availability
- Reliability
- Security
- Accuracy (of measurement, operation)
- Speed (of operation, service)
- PowerDrain (of service execution, of operation)
- Cost

The dynamism of the Latency and Speed can be calculated with the MessageProbe (also called IPSniffer for historical reasons) ontology. As shown in rule *MessageCallRelationship*, it calculates the round trip calling for a service, service execution time (speed of service), and build the invocation relationships between processes. This rule first retrieves all the messages that are supposed to be a complete round trip call from a client to the service, then calculates the related information using SWRL built-in functions.

Rule: MessageCallRelationship

$ipsniffer : messageID(?message1, ?messageid) \wedge$
$ipsniffer : messageID(?message2, ?messageid) \wedge$
$ipsniffer : messageID(?message3, ?messageid) \wedge$
$ipsniffer : messageID(?message4, ?messageid) \wedge$
$abox : hasURI(?message1, ?u1) \wedge$
$abox : hasURI(?message2, ?u2) \wedge$
$abox : hasURI(?message3, ?u3) \wedge$
$abox : hasURI(?message4, ?u4) \wedge$
$swrlb : containsIgnoreCase(?u1, "clientbegin") \wedge$
$swrlb : containsIgnoreCase(?u2, "servicebegin") \wedge$
$swrlb : containsIgnoreCase(?u3, "serviceend") \wedge$
$swrlb : containsIgnoreCase(?u4, "clientend") \wedge$
$ipsniffer : messageSourceIP(?message1, ?ip1) \wedge$
$ipsniffer : ipaddr(?ip1, ?ipa1) \wedge$
$ipsniffer : ipaddr(?ip2, ?ipa2) \wedge$
$ipsniffer : hasMessage(?process1, ?message1) \wedge$
$ipsniffer : hasProcessID(?process1, ?pid1) \wedge$
$ipsniffer : messageTargetIP(?message1, ?ip2) \wedge$
$ipsniffer : initiatingTime(?message1, ?time1) \wedge$
$ipsniffer : messageSourceIP(?message2, ?ip3) \wedge$
$ipsniffer : messageTargetIP(?message2, ?ip4) \wedge$
$ipsniffer : ipaddr(?ip3, ?ipa3) \wedge$
$ipsniffer : ipaddr(?ip4, ?ipa4) \wedge$
$ipsniffer : messageTargetPort(?message2, ?port2) \wedge$
$ipsniffer : hasMessage(?process2, ?message2) \wedge$
$ipsniffer : hasProcessID(?process2, ?pid2) \wedge$
$ipsniffer : initiatingTime(?message2, ?time2) \wedge$
$ipsniffer : messageSourceIP(?message3, ?ip5) \wedge$
$ipsniffer : messageTargetIP(?message3, ?ip6) \wedge$
$ipsniffer : ipaddr(?ip5, ?ipa5) \wedge$
$ipsniffer : ipaddr(?ip6, ?ipa6) \wedge$
$ipsniffer : messageTargetPort(?message3, ?port3) \wedge$
$ipsniffer : hasMessage(?process3, ?message3) \wedge$
$ipsniffer : hasProcessID(?process3, ?pid3) \wedge$
$ipsniffer : initiatingTime(?message3, ?time3) \wedge$
$ipsniffer : messageSourceIP(?message4, ?ip7) \wedge$
$ipsniffer : messageTargetIP(?message4, ?ip8) \wedge$
$ipsniffer : ipaddr(?ip7, ?ipa7) \wedge$
$ipsniffer : ipaddr(?ip8, ?ipa8) \wedge$
$ipsniffer : messageTargetPort(?message4, ?port4) \wedge$
$ipsniffer : hasMessage(?process4, ?message4) \wedge$
$ipsniffer : hasProcessID(?process4, ?pid4) \wedge$
$ipsniffer : initiatingTime(?message4, ?time4) \wedge$

$temporal : duration(?d1, ?time1, ?time4, temporal : Milliseconds) \wedge$
$temporal : duration(?d2, ?time2, ?time3, temporal : Milliseconds) \rightarrow ipsnif fer : invoke(?message1, ?message2) \wedge$
$sqwrl : select(?ip1, ?ipa1, ?pid1, ?ipa2, ?port2, ?pid2, ?d1, ?d2)$

QoS ontology (actually it is a set of ontologies) formally defines the above important QoS parameters. It also contains properties for these parameters, such as its nature (dynamic, static) and the impact factor. There is also a *Relationship* concept in order to model the relationships between these parameters. The QoS ontology is developed based on the Amigo QoS ontology (IST Amigo Project, 2006) and OWL-Q ontology (Kritikos and Plexousakis, 2007). It simplified the OWL-Q ontology in which we adopted its QoS specification idea, and included our listed parameters.

QoS ontologies include a QoSSpec ontology which defines QoS offers and requests as required by SLA. A QoSSelection class is used to define a list of <metric, weight> entries, in which a web service requester can weight the metrics of his interest. QoSSpec is used to specify the actual QoS description of a web service. The QoSSpec ontology is shown in Figure 6.3. Another important ontology is the QoSMetric ontology as shown in Figure 6.4. It defines all the network performance parameters used to measure the quality of a network, and other parameters listed in the beginning of this section. It also defines the functions used to calculate a metric, including the boolean functions, aggregation functions, and arithmetic functions.

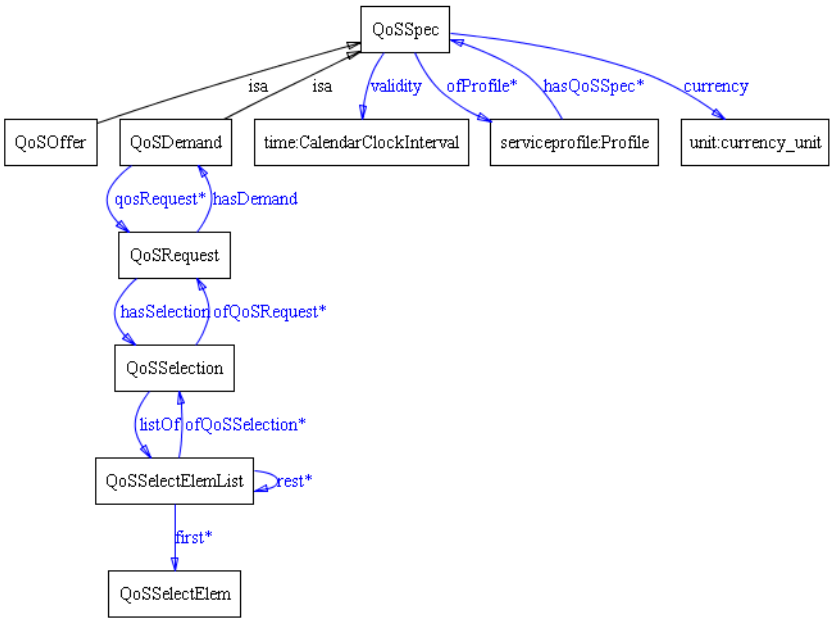

Figure 6.3: QoSSpec ontology (Partial)

6.4 Software architecture ontologies design

6.4.1 Semantic architectural styles

The semantic models for popular architectural styles are developed based on published work. For example, the models for a Client-Server style, a Pipe-Filter style are based

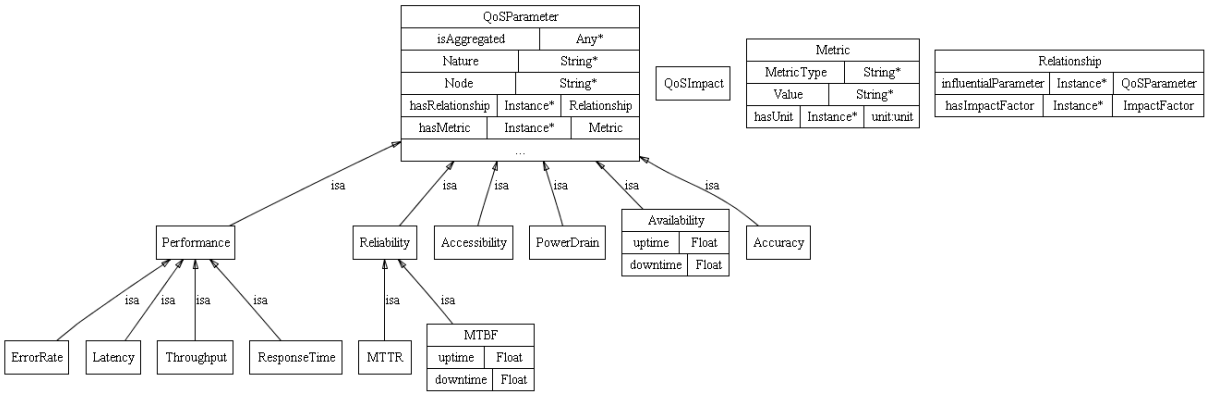

Figure 6.4: QoSMetric ontology (Partial)

on (Garlan et al., 2000), a Publish-Subscribe style, and a Peer-to-Peer style are based on (Clements et al., 2003), and other work on software architectural styles (Shaw and Clements, 1997) (Bhattacharya and Perry, 2005) in which topology constraints are clearly specified. In general, the ACME (Garlan et al., 2000) specification for an architectural style is developed first, and then the specification is transformed into the OWL model.

One of the important issues for semantic architecture modeling is to model software architecture constraints, which are design decisions behind the rationale of architectural styles. There can be three different types of architecture constraints:

- *Static structural parts in an architectural style*. This will make sure that an architectural style has correct types of components, connectors, roles, and ports. For example, a *Repository Style* should have at least one *Repository* component which has port *Provide*, at least one *Access* connector that has roles *Provider* and *User*.
- *Dynamic connections between structural parts, including topological constraints (Bhattacharya and Perry, 2005) (Shaw and Clements, 1997)*. That is to say, how a component, connector, role, and port are connected with each other. For example, the *Repository Style* has a “Star” control topology and data topology, the port *Provide* in the *Repository* component should be connected to the role *Provider* in the *Access* connector.
- *Externally visible properties of the structural parts* (Bass et al., 2003a). This includes some quality attributes, such as performance, fault tolerance, feature combination constraints, and also some functional properties, for example the required software and hardware platforms to run a component.

OWL-DL is strong in modeling vocabularies and their relationships in a formal way. It provides good capabilities for specifying the first constraints where architecture vocabularies are modeled as concepts and/or properties. Additionally, component and connectors can be annotated with functional and nonfunctional properties to resolve the third type of constraints. There are also cardinality and universal/existential constraint constructs in OWL-DL to specify constraints for the second type of constraints, mainly local constraints. To specify global constraints and other types of constraints that are beyond the capabilities of OWL-DL, we are applying SWRL to enhance constraint specification capabilities.

Additionally, when we build the ontology for architectural styles, their quality attributes and application domains as classified in (Niemelä et al., 2005) are considered in order to facilitate a designer in choosing the right style to achieve the required quality requirements. For example, the Pipe-Filters are used in situations where data stream processing is needed, and has *Simplicity*, *Reusability*, and *Maintainability* as its quality attributes.

According to the definition and nature of an architectural style, it can be considered as a whole-part relationship with its composed components, connectors, ports, roles, and properties. Therefore an architectural style can be modeled with whole-part relationship modeling using OWL[11].

List: architectural style using OWL-DL

$ArchitecturePart \sqsubseteq \top$
$Component \sqsubseteq ArchitecturePart$
$Connector \sqsubseteq ArchitecturePart$
$Port \sqsubseteq ArchitecturePart$
$Role \sqsubseteq ArchitecturePart$
$Property \sqsubseteq ArchitecturePart$
$\top \sqsubseteq \forall hasPort.Port$
$\top \sqsubseteq \forall hasRole.Role$
$\top \sqsubseteq \forall hasProperty.Property$
$\top \sqsubseteq \forall hasArchitecturePart.ArchitecturePart$
$\top \sqsubseteq \forall hasComponent.Component$
$\top \sqsubseteq \forall hasConnector.Connector$
$hasComponent \sqsubseteq hasArchitecturePart$
$hasConnector \sqsubseteq hasArchitecturePart$
$ArchStyle \sqsubseteq \top$
$ArchStyle \sqsubseteq \neg ArchitecturePart$
$ArchStyle \sqsubseteq \forall hasArchitecturePart(Component \sqcup Connector \sqcup Port \sqcup Role \sqcup Property)$

The following is the specification for a Publish-Subscribe style, in which there are three types of components: Publisher with port Publish, Subscriber with port Subscribe and PubSubCore with port Notify, and uses Event as the connector, in which we do not model its role. There should be at least one port for each component. The Event connector is the same class as in the AtomicConnector ontology, and may need an additional Distributor connector to work in distributed computing environments

List: Publish-Subscribe style using OWL-DL

$PubSubComponent \sqsubseteq Component$
$Publisher \sqsubseteq PubSubComponent$
$Subscriber \sqsubseteq PubSubComponent$
$PubSubCore \sqsubseteq PubSubComponent$
$PubSubComponent \equiv (PubSubCore \sqcup Subscriber \sqcup Publisher)$
$PubSubEvent \sqsubseteq Connector$
$PubSubProtocol \sqsubseteq PubSubConnector$
$Event \equiv PubSubProtocol$
$PubSubPort \sqsubseteq Port$
$Notify \sqsubseteq PubSubPort$
$Publish \sqsubseteq PubSubPort$
$Subscribe \sqsubseteq PubSubPort$
$Publisher \sqsubseteq \exists hasPort.Publish$
$Publisher \sqsubseteq \geqslant 1 hasPort$
$Subscriber \sqsubseteq \exists hasPort.Subscribe$
$Subscriber \sqsubseteq \geqslant 1 hasPort$
$PubSubCore \sqsubseteq \exists hasPort.Notify$
$PubSubCore \sqsubseteq \geqslant 1 hasPort$
$PublishSubscribe \sqsubseteq ArchStyle$
$PublishSubscribe \sqsubseteq \forall hasArchitecturePart($
$PubSubComponent \sqcup PubSubConnector$
$\sqcup PubSubPort \sqcup PubSubRole \sqcup Property)$

[11] http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/index.html

6.4.2 Semantic OSGi Components

The OSGi component ontology is based on the OSGi's Declarative Service specification (OSGi Alliance, 2007). It specifies the *Component* (as a concept) dynamic status, for example whether it is *enabled*, and also static characteristics such as its *reference* to other service, its *implementation* interface, and *service*s provided. As we are going to illustrate the SWRL rules using OSGi component as examples, Figure 6.5 shows partially the details of the *OSGi* component ontology for facilitating understanding of SWRL rules developed for OSGi components validation.

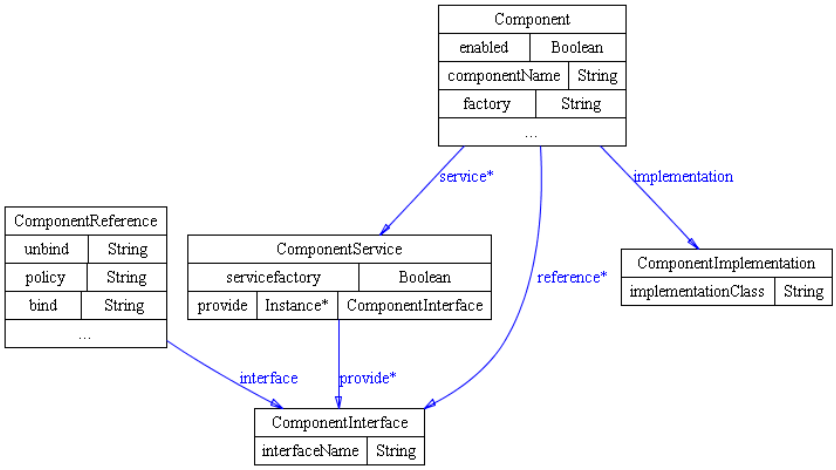

Figure 6.5: OSGi component ontology (Partial)

We adopted OWL-S 1.2 Pre-Release[12] as the basis for the semantic models for traditional component services and web services, because OWL-S is a de facto standard for semantic web service. The ServiceProfile ontology can be used to categorize services for better service matching. Services are modeled as processes (in the Process ontology) in which input, output, precondition, and results are used to describe a service. An atomic process models an action a service can perform in a single interaction, whereas a simple process is an abstraction mechanism to provide multiple views of the same process. Having a look at the component models of OSGi, every operation in a component can be modeled as an atomic process, and then a simple process for the whole component is composed of these atomic processes using the *realizedBy* object property in the Process ontology.

Now we exemplify how to use the Process ontology to model component services. Table 6.4 shows component dependencies for the Limbo pervasive service compiler (Hansen et al., 2008), following a Repository style. There is one Repository component and four Repository clients. The Limbo Repository component has five operations which are defined as five atomic processes in the Process ontology, where the return types are modeled as process *Output*s, and method signatures are modeled as *Input*s. An instance of *SimpleProcess* is defined which is composed with these five *AtomicProcesses* correspondingly. In these Repository Client components, instances of *SimpleProcess* for each operation are defined in a similar way. If there is a reference from the Repository Client to the Repository component in a method, the atomic process for this method will have a *Participant* instance which should be the *Repository* component. The inputs of this atomic process will contain

[12] http://www.ai.sri.com/daml/services/owl-s/1.2/

both the signature of the referenced operation together with its return type, and method signature of itself. A rule called *Rule: check_OSGi_Reference_Details* is show in Section 8.3.2 is an example to show how this works.

Repository operations	Limbo	UPnP	State Machine	Probe
URI getOntologyURI()		Y	Y	
Definition getWSDL()		Y		
File getWSDLFile()	Y	Y		
HashMap<String,String> getLimboConfiguration()	Y	Y		Y
URI getHydraOntologyExtension (File wsdlFile)	Y			

Table 6.4: Limbo component references

In this way, all component models are enhanced with capabilities for knowing their implementation details, which provides a unified way for modeling the details of component services, including service dependencies among components, which can be used at runtime to conduct validations of architectural styles and configurations. This way of modeling component details is the key to find the correct component when there are multiple components where interfaces should be matched.

An important benefit of these semantic component models are that they are clarifying the relationships between some of the concepts for different component models, which can facilitate the usage and understanding of these component models. For example, the concept of OSGi component *Reference* is a subclass of *ComponentRequiredService*, and Fractal[13] *Binding* is a subclass of *ComponentReference*.

6.4.3 Semantic Connectors

Although there are a number of classifications for software connectors in literature, the one from Mehta (Mehta et al., 2000) is arguably the most accepted and provides a comprehensive list of connectors, which is used as the basis for our semantic connector model. All the eight type of connectors, including Procedure Call, Event, Linkage, Distributor, Arbitrator, Stream, Data Access, and Adaptor are modeled in the AtomicConnector ontology as shown in Figure 6.6.

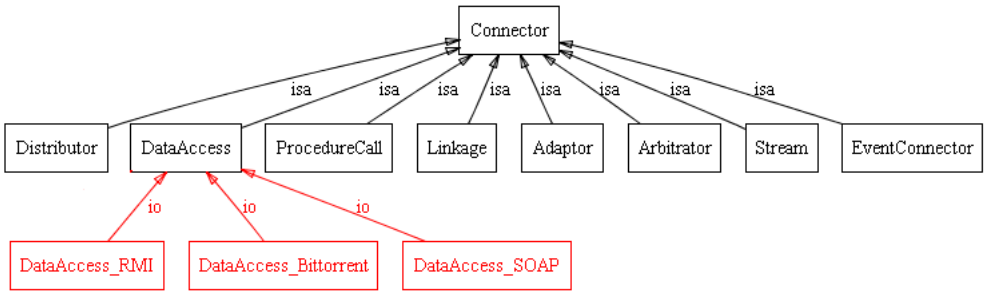

Figure 6.6: Atomic connector model (Partial)

Connectors can be composed to form a composite connector, for example, 13 different types of distribution connectors are explored for highly distributed data intensive systems

[13] http://fractal.objectweb.org/

by Mattmann (Mattmann, 2007). In his dissertation, details on how these connectors are composed with properties are specified in XML DTDs. We encode all this knowledge in our connector models. For the modeling of a composite connector where the order of connectors matters, we use OWL sequences as proposed in (Drummond et al., 2006). From (Mattmann, 2007), we can say that a basic distribution connector is composed with three connectors in this order: Data Access, Stream, and Distributor. This is a basic distribution connector, and a SOAP[14] connector is one specialization of it where a SOAP procedure call is first involved.

In practice, connectors are not always a first class citizen in design and implementation. Therefore connectors may not easily be spotted and may be mixed with components, and are not easily separated from components. In this case, we will allow that components are directly connected with other, rather than through an artificial connector, which may unnecessarily make the understanding and modeling more complicated.

Now we can apply the these models to practical usage. The Hydra Event Manager (as introduced briefly in (Zhang and Hansen, 2008a)) is using a Publish-Subscribe architectural style, and it has three typical components for this style as introduced in Section 6.4.1, and is using Event as connector with SOAP as the distribution and transportation connector. This is summarized in Table 6.5. Typical description logic reasoning can then be utilized to check whether this model for the Hydra Event Manager is consistent or not using RacerPro[15], and it turns out to be consistent.

architectural style	Publish-Subscribe
Component model	Web service
Component-Port	EMPub - publish EMSub - subscribe EMCore - notify
Connector-Role	EMEvent - publisher, subscriber EMSOAP - transportpub, transporsub

Table 6.5: Hydra Event Manager model

6.5 Dynamic context modeling in SeMaPS ontologies

As said in the beginning of this chapter, it is very important to know dynamism of the underlying pervasive system for achieving self-management. The dynamism includes the following:

- Device and system run time status. It is a common sense that embedded devices are designed and running as state machines, therefore we could make use of this idea for achieving self-management. For example, self-diagnosis according to the states of devices.
- Service calling relationships. It is important to know whether certain devices are available or not, which could be achieved by initiate a service call to a device and detecting whether its response time is within the scope of expectation or not. It is also beneficial to know a system run time architecture in order to decide whether the system is running in a way following certain architecture constraints. The building of run time architecture could be achieved by monitoring service calling relationships.
- Components run time states. Components and connectors are the underlying software unit to implement services. They should follow the specified architecture con-

[14] http://www.w3.org/TR/soap/

[15] RacerPro version 1.9.2Beta. http://www.racer-systems.com/

straints, and also we could change the system configuration by loading/unloading or enabling/disenabling components/connectors.

To model device state changes, a StateMachine ontology is developed based on (Dolog, 2004), with many improvements to facilitate self-management: the *State* concept has data-type property *isCurrent* to indicate whether a state is current or not for the purpose of device monitoring, a *doActivity* object property is added to the *State* in order to specify the corresponding activity in a state, and also a data-type property *hasResult* is added to the *Action* (including activity) concept in order to check the execution result at runtime, together with three extra data-type properties to model historical action results in order to conduct history based self-management.

To model the invocation of services, a MessageProbe ontology is developed to model the monitoring the liveness of a computing node, and to facilitate the monitoring of QoS, such as the request/response time of a corresponding service call. The *SocketProcess* concept is used to model a process running in a client or service, and *SocketMessage* to model a message sent between client and service. There is also a concept called *IPAddress*, which is related to *Device* with a property *hasIPAddress* in the Device ontology. The object properties *invoke*, *messageSourceIP*, and *messageTargetIP* are used to build the invoking relationships, and data type property *initiatingTime* is used to model the time stamp for a message.

The key to view a software component and connector in a configuration for a pervasive system is to understand the services a component and connector can provide and which they require in order to function. This is modeled by a Service ontology (using a service profile) and a ProcessModel ontology (using a service process model), the same idea as in OWL-S[16] ontologies for semantic web services. The Service ontology and ProcessModel ontology are used at runtime to conduct self-configuration and self-adaptation, potentially based on quality-of-service requirements (supported by the QoS ontology).

In summary, the dynamic contexts are modeled with runtime concepts and properties in the related ontologies, mainly the StateMachine ontology, MessageProbe ontology, Malfunction ontology, Component ontology, QoS ontology, Service and ProcessModel ontology, and other concepts and properties in the Device ontology, such as *currentMalfunction* and *System*. The *currentMalfunction* will be used to store the current diagnosis information for the malfunction case, *System* is used to dynamically model a device joining and leaving and to reflect the composition of a system.

6.6 Complex context specification with SWRL rules

SWRL can be used to develop complex contexts (Zhang and Hansen, 2008b). For scenario 1 in Chapter 3, We can specify a *farAwayFromHome* context (e.g. 100 miles away from home using the GPS distance calculation formula[17]). Then this new context can be used to take actions, for example, if the GPS location of Smith is 100 miles away from his home, his home surveillance system will switch automatically to the highest security level with all cameras turned on.

Rule: FarAwayFromHome
$person : hasHome(?person, ?home) \wedge$
$person : inLocation(?person, ?coord1) \wedge$

[16] http://www.w3.org/Submission/OWL-S/

[17] How to calculate the distance between two points on the Earth. http://www.meridianworlddata.com/Distance-Calculation.asp

$loc : hasCoordinates(?home, ?coord2) \wedge$
$coord : latitude(?coord1, ?lan1) \wedge$
$coord : latitude(?coord2, ?lan2) \wedge$
$swrlb : subtract(?sub1, ?lan1, ?lan2) \wedge$
$swrlb : multiply(?squaresublan, ?sub1, ?sub1) \wedge$
$swrlb : multiply(?par1, ?squaresublan, 4774.81) \wedge$
$coord : longitude(?coord1, ?long1) \wedge$
$coord : longitude(?coord2, ?long2) \wedge$
$swrlb : subtract(?sub2, ?long1, ?long2) \wedge$
$swrlb : multiply(?squaresublong, ?sub2, ?sub2) \wedge$
$swrlb : multiply(?par2, ?squaresublong, 2809) \wedge$
$swrlb : add(?parameter, ?par1, ?par2) \wedge$
$swrlm : sqrt(?distance, ?parameter) \wedge$
$swrlb : greaterThan(?distance, 100) \wedge$
$\rightarrow sqwrl : select(?person, ?home, ?distance) \wedge$
$farAwayFromHome(?person, \text{“}true\text{”})$

7 Self-management rules based on SeMaPS ontologies

7.1 Self-diagnosis rules

Monitoring and diagnosis rules are the basis for the diagnosis service and can be developed based on the introduced SeMaPS ontologies. Besides the rules shown in deliverable D4.3 (Ingstrup et al., 2008), we have developed more rules for self-diagnosis, for example the following rule for wind meter (used in the weather state prototype developed by CNet) diagnosis:

Rule: Windmeter_PowerDown

$$
\begin{aligned}
&device : Windmeter(?device) \wedge \\
&device : hasStateMachine(?device, ?statemachine) \wedge \\
&statemachine : hasStates(?statemachine, ?state) \wedge \\
&statemachine : doActivity(?state, ?action) \wedge \\
&statemachine : actionResult(?action, ?result) \wedge \\
&statemachine : actionResultTimestamp(?action, ?time) \wedge \\
&statemachine : historicalResult1(?action, ?result1) \wedge \\
&statemachine : historicalResult2(?action, ?result2) \wedge \\
&statemachine : historicalResult3(?action, ?result3) \wedge \\
&statemachine : timeStampResult3(?action, ?time3) \wedge \\
&swrlb : subtract(?temp1, ?result, ?result1) \wedge \\
&swrlb : subtract(?temp2, ?result1, ?result2) \wedge \\
&swrlb : subtract(?temp3, ?result2, ?result3) \wedge \\
&swrlb : abs(?a1, ?temp1) \wedge \\
&swrlb : abs(?a2, ?temp2) \wedge \\
&swrlb : abs(?a3, ?temp3) \wedge \\
&swrlb : add(?a, ?a1, ?a2) \wedge \\
&swrlb : add(?b, ?a, ?a3) \wedge \\
&temporal : duration(?dura, ?time, ?time3, temporal : Seconds) \wedge \\
&swrlb : equal(0, ?b) \wedge \\
&swrlb : greaterThanOrEqual(?dura, 30) \rightarrow sqwrl : select(?b) \wedge \\
&sqwrl : select(?time, ?time3, ?dura) \wedge \\
&device : currentMalfunction(?device, error : PowerDown) \wedge \\
&device : currentMalProbability(error : PowerDown, \text{"90\%"})
\end{aligned}
$$

If within 30 seconds, the measured value for wind meter remains the same, then most probably the power supply for the wind meter is down, with the probability of 90%.

7.2 Self-configuration rules

To resolve the problem that often causes the ventilating system down in Smith's home in scenario 1 of Chapter 3, a third party service component is found online when the SmartHome system is trying to search for a solution. The following rule is used to detect interface mismatch and resolve this mismatch by adding a connector to the current configuration.

Here in order to simplify the writing of rules (the reason being that current Protege SWRL APIs can not parse rdf:literal as in the OWL-S 1.2 pre-release), we changed the range of the data type property *parameterValue* to *xsd:string* defined in the OWL-S Process ontology. To correctly justify a reference, the component package and component name, method name, method signature including data type and order, and return type, should be consistent in both referencing and referenced components. Using the service details as provided by the ServiceProfile and Process ontologies, we can then retrieve the details of the services and its method signatures.

For an OSGi component, if it has a reference which has a cardinality of the form "1." (at least one reference to other services), then there must be a component providing that required service. In a referencing component, the references to another component is modeled in the *hasInput* datatype property in the *ProcessModel* ontology, in the format "*component name(including package name)+operation name#input types with orders$return type*". Then this information is compared with that from the referenced component with respect to a specific interface. If they are not exactly matched, then the references are invalid. Additionally, if a connector resolving the interface mismatch exists (its instance is *adaptConnector1* as in the *OSGiComponent* ontology), we can then infer that an adaptor connector needs to be added to the current configuration in order to correctly match the references, shown in the last line of the rule.

Rule: OSGi_InterfaceMismatchResolution

$CurrentConfiguration(?con) \wedge$
$hasComponent(?con, ?comp1) \wedge$
$osgi : componentName(?comp1, ?compname1) \wedge$
$osgi : reference(?comp1, ?ref1) \wedge$
$osgi : cardinality(?ref1, ?car1) \wedge$
$swrlb : containsIgnoreCase(?car1, \text{"1."}) \wedge$
$osgi : interface(?ref1, ?inter1) \wedge$
$osgi : interfaceName(?inter1, ?name1) \wedge$
$hasComponent(?con, ?comp2) \wedge$
$architectureRole(?comp2, ?role2) \wedge$
$osgi : service(?comp2, ?ser2) \wedge$
$osgi : provide(?ser2, ?inter2) \wedge$
$osgi : interfaceName(?inter2, ?name2) \wedge$
$osgi : componentName(?comp2, ?compname2) \wedge$
$swrlb : equal(?name1, ?name2) \wedge$
$component : componentServiceDetails(?comp1, ?pr1) \wedge$
$service : presents(?pr1, ?prservice1) \wedge$
$profile : has\_process(?prservice1, ?process1) \wedge$
$process : realizedBy(?process1, ?aprocess1) \wedge$
$process : hasInput(?aprocess1, ?input1) \wedge$
$process : parameterValue(?input1, ?ivalue1) \wedge$
$component : componentServiceDetails(?comp2, ?pr2) \wedge$
$service : presents(?pr2, ?prservice2) \wedge$
$profile : has\_process(?prservice2, ?process2) \wedge$
$process : realizedBy(?process2, ?aprocess2) \wedge$
$process : hasInput(?aprocess2, ?input2) \wedge$
$process : name(?aprocess2, ?proname2) \wedge$
$process : hasOutput(?aprocess2, ?proout2) \wedge$
$process : parameterValue(?input2, ?ivalue2) \wedge$
$process : parameterValue(?proout2, ?ovalue2) \wedge$
$swrlb : stringConcat(?str1, ?compname2, \text{" + "}) \wedge$
$swrlb : stringConcat(?str2, ?str1, ?proname2) \wedge$
$swrlb : stringConcat(?str3, ?str2, \text{"\#"}) \wedge$
$swrlb : stringConcat(?str4, ?str3, ?ivalue2) \wedge$
$swrlb : stringConcat(?str5, ?str4, \text{"\$"}) \wedge$
$swrlb : stringConcat(?str6, ?str5, ?ovalue2) \wedge$
$swrlb : equal(?ivalue1, ?str6) \wedge$
$swrlb : substringBefore(?temp1, ?ivalue1, \text{" + "}) \wedge$
$swrlb : equal(?temp1, ?compname2) \wedge$
$swrlb : substringAfter(?temp, ?ivalue1, \text{" + "}) \wedge$
$swrlb : substringBefore(?op, ?temp, \text{"\#"}) \wedge$
$swrlb : equal(?op, ?proname2) \wedge$
$swrlb : substringAfter(?temp2, ?ivalue1, \text{"\#"}) \wedge$
$swrlb : substringBefore(?inputtype, ?temp2, \text{"\$"}) \wedge$
$swrlb : notEqual(?inputtype, ?inputtype2) \wedge$
$swrlb : substringAfter(?returntype, ?ivalue1, \text{"\$"}) \wedge$
$swrlb : equal(?ovalue2, ?returntype)$
$\rightarrow sqwrl : selectDistinct(?comp1, ?comp2, ?inputtype, ?inputtype2)$
$\wedge sqwrl : select(\text{"Input type mismtach: invalid references"})$
$\wedge sqwrl : select(\text{"An adaptor connector is Needed"})$
$\wedge hasComponent(?con, adaptConnector1)$

This inferred results are then published to the event manager, and listened by the ASL-host service, to bind the "AdaptConnetor1" into the current configuration.

7.3 Self-adaptation rules

In corresponding to the identified scenarios of case 3, we design the following rule to adapt the system to choose wired connection as the underlying Internet service. The rule is as followed in which we are making use of the QoS ontology set, in which we do not need to consider the cost of the service as it is not important in this situation.

Rule: SpeedReliabiltyPriority
$hasPerformance(?net1, ?transfer1) \wedge$
$hasReliability(?net1, ?r1) \wedge$
$QoS : hasMetric(?transfer1, ?metric1) \wedge$
$QoS : Value(?metric1, ?value1) \wedge$
$hasPerformance(?net2, ?transfer2) \wedge$
$QoS : hasMetric(?transfer2, ?metric2) \wedge$
$QoS : Value(?metric2, ?value2) \wedge$
$hasReliability(?net2, ?r2) \wedge$
$swrlb : greaterThan(?value1, ?value2) \wedge$
$swrlb : greaterThan(?r1, ?r2)$
$\rightarrow sqwrl : select(?net1, ?net2, ?transfer1, ?metric1, ?value1)$
$\wedge\, isCurrent(?net1, "true")$

This rule selects the underlying network that has better connection, at the same time higher reliability. And then this rule will make the selected network as the current with by setting its *isCurrent* property to “true”. In reality, as there may be conflicting QoS requirements and priority scheme, SWRL rules itself could not achieve the resolution of the conflicting choices. In this case, we need a planning mechanism to resolve such a situation, which is detailed in Chapter 8.

7.4 Architectural styles and configurations validation at runtime

Due to the open and dynamic nature of pervasive computing and Internet scale computing, software components providing services can join and leave a system at anytime, hence component configurations can be changed more often than those running in closed world environments. This naturally requires that these new configurations should be valid and still follow certain architectural styles. That is to say, we should dynamically validate component configurations and architectural styles.

We assume that components and connectors are annotated with their roles in an architectural style. It does not matter which and how many roles they can play. This can be achieved at design time to designate the nature of a component and connector. For example, in a Repository style, a Repository component and Repository Client component. We also assume that details of components (for example, following the specification of a certain component model) and connectors (for example, following the details as classified in (Mehta et al., 2000) (Mattmann, 2007)) are known when they are going to be validated for certain configuration. This may seemingly contradict to the OWA, but it is reasonable as a starting point towards an open world software architecture. As component and connectors may change at runtime, we would check that the new configuration is still following a designated architectural style. Therefore, if there are the required components and connectors, the number of component/connectors are meeting constraints and other requirements, and the components/connectors are correctly referenced with each other, then these components are following an architectural style.

Figure 7.1 shows a general approach for architectural styles and configurations validation using SWRL rules. Firstly, there should be components/connectors corresponding to

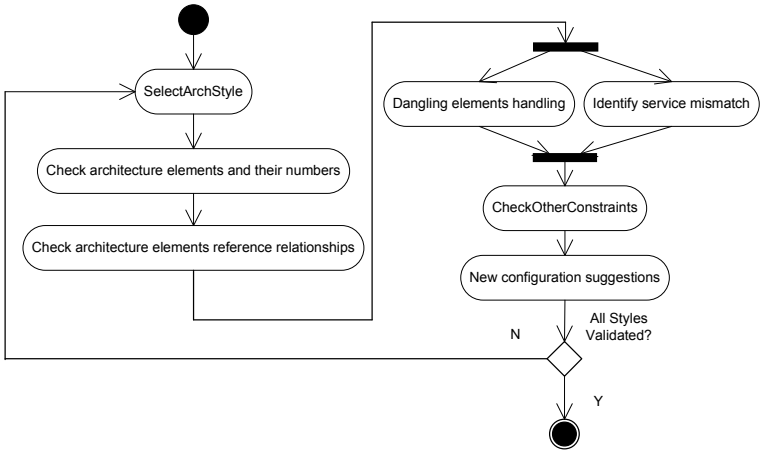

Figure 7.1: Architecture validation activities

an architectural style (maybe connectors are not used explicitly for some situations). This will ensure that a specific configuration following an architectural style has the corresponding architecture elements. The total number of the component/connectors should be the same as these ones under consideration (or larger when a component/connector plays multiple architecture roles if simply sum the figures got from SWRL queries for the components/connectors numbers).

Then these components/connectors need to be correctly referenced by each other. For example, a Repository client should be connected to a Repository component, not necessarily directly connected to, but possibly with a connector as an intermediate element. This will also make sure that the correct topology will be respected (in the Repository style, it is a “Star” topology as said). If there is a Repository that has no reference(s), then it is a dangling component, which may need to be deactivated. In OSGi, this can be achieved by setting its *enabled* property to “false”. If there is only one Repository component, and it is a dangling component, then it is an invalid configuration. As component services may not be exactly matched, for example interface signature mismatch, the mismatches should be identified according to the semantic details of components and services.

Some other property constraints for an architectural style should be checked, such as quality of service requirements, software/hardware platform constraints, and other contextual constraints. For example, some feature combination may not be valid, for example, an OSGi server on a JME platform may be not a valid combination at present stage.

Using the Equinox DS service, we can easily get the whole list of component/connector instances to be validated. After checking the valid references, invalid references, and possible mismatches between components/connectors, then we can get a list of components/connectors for a valid configuration. If there are mismatches, we can get some suggestions to resolve the mismatches through rule inferring, and may help to add new components/connectors into the current configuration.

7.5 Probability handling in diagnosis

7.5.1 Survey on probability in semantic web

When conducting diagnosis, it is not always possible to ascertain that an error is 100% comes from a certain source, it is an uncertain situation in essence. The existing semantic web languages are based on classical logic which is inadequate to represent uncertainty. Therefore, quite some research remains to be done in in improvving the semantic web technologies' ability to handle uncertainty.

PR-OWL (COSTA and LASKEY, 2006) is an OWL ontology for describing first order probabilistic models, which models Multi-Entity Bayesian Networks (MEBN) that define probability distributions over first-order theories in a modular way. MEBN is supposed to be capable of representing and reasoning about probabilistic information about any sentence in first-order logic by compiling it into a Bayesian Network. However, as the connection between a statement in PR-OWL and a statement in OWL is not formalized, it is unclear how to perform the integration of ontologies that contain both formalisms (Predoiu and Stuckenschmidt, 2008).

BayesOWL (Ding et al., 2006) is a probabilistic extension of propositional logic which allows including probabilistic mappings between different ontologies into the inference procedure. It is an approach for representing probabilistic information about class membership within OWL ontologies, and not probabilistic expression based on properties. Therefore its big limitation is that it can not represent probabilistic information on any relations except the subsumption relation. Therefore we may conclude that PR-OWL has better expressive capabilities than BayesOWL.

There are many others approaches for example extending RDF as surveyed in (Predoiu and Stuckenschmidt, 2008). For the Hydra purpose, we need to make the probabilistic capabilities work on the rule level, therefore neither of the current approaches are suitable for us. There is a work on adding fuzzy reasoning capabilities to SWRL called f-SWRL (Pan et al., 2005). f-SWRL is designed to improve the situation that SWRL fails at representing vague and imprecise knowledge and information. f-SWRL can be used in diagnosis to express the probability, but the problem of it is that it remains at the theoretical study phase and no tools are available to support this.

Therefore for the moment we propose a simple solution to build directly the probability into SWRL rule by a datatype property *currentMalProbability* as illustrated in the following rule. This approach is also adopted by Amigo project.

7.5.2 Diagnosis with probability

In the agriculture scenarios, we need to monitor that the flowing of the feeding of food mixed in some silos using pressure sensors and light sensors. The pressure sensor detects the speed of the silos with original food material, and the light sensor is used to monitor whether the food is pumped into a feeding silo which should detect that the light is dark when the pumping process started. Otherwise, there is something wrong (with the following probabilities) as shown in the rule. In this situation, the possibility of pipe broken is "70%", and the possibility of silo broken is much less which is "30%".

Rule: Silo_DiagnosisProbability
$device : PressureMeter(?device) \wedge$
$device : hasStateMachine(?device, ?statemachine) \wedge$
$statemachine : hasStates(?statemachine, ?state) \wedge$
$statemachine : doActivity(?state, ?action) \wedge$
$statemachine : actionResult(?action, ?result) \wedge$

$$
\begin{aligned}
&statemachine : historicalResult1(?action, ?result1) \wedge \\
&statemachine : historicalResult2(?action, ?result2) \wedge \\
&statemachine : historicalResult3(?action, ?result3) \wedge \\
&swrlb : add(?tempaverage, ?result1, ?result2, ?result3) \wedge \\
&swrlb : divide(?average, ?tempaverage, 3) \wedge \\
&swrlb : subtract(?temp1, ?result, ?result1) \wedge \\
&swrlb : subtract(?temp2, ?result1, ?result2) \wedge \\
&swrlb : subtract(?temp3, ?result2, ?result3) \wedge \\
&swrlb : add(?temp, ?temp1, ?temp2, ?temp3) \wedge \\
&swrlb : lessThan(?temp, 0.0) \wedge \\
&device : LightSensor(?device2) \wedge \\
&device : hasStateMachine(?device2, ?statemachine2) \wedge \\
&statemachine : hasStates(?statemachine2, ?state2) \wedge \\
&statemachine : doActivity(?state2, ?action2) \wedge \\
&statemachine : actionResult(?action2, ?result22) \wedge \\
&abox : isNumeric(?result22) \wedge \\
&swrlb : lessThan(?result22, 45) \rightarrow device : currentMalfunction(device : Pipe1, error : PipeBroken) \wedge \\
&device : currentMalProbability(error : PipeBroken, "70\%") \wedge \\
&device : currentMalfunction(device : Silo1, error : SiloBroken) \wedge \\
&device : currentMalProbability(error : SiloBroken, "30\%") \wedge \\
&sqwrl : select("silodiagnosis")
\end{aligned}
$$

7.6 Discussion

SeMaPS ontologies cover a relatively complete set of knowledge for self-management, including all dynamic information of pervasive systems, for example runtime device states. SWRL is very flexible and powerful to develop self-management rules based on the SeMaPS ontologies. The development of OWL/SWRL ontologies is supported by Protege[1]. A developer will find it relatively easy to develop SWRL rules as the user interface for SWRL rule development is user-friendly.

[1] protege homepage. http://protege.stanford.edu/ and Protege-SWRL tab: http://protege.cim3.net/cgi-bin/wiki.pl?SWRLTab

8 Semantic software architecture enabled and QoS based planing

8.1 Survey on planning techniques for the Goal management layer in pervasive systems

It has shown that the traditional AI planning techniques could be used in pervasive computing for self-management purposes (Arshad et al., 2003)(Arshad et al., 2007; Ranganathan and Campbell, 2004), in which STRIPS-planning is used as policy engine to generate plans. It was pointed out in (Alia et al., 2006) that this approach will not scale well, especially in open systems. From the performance figures in (Arshad et al., 2007), we can see that the planning performance is not very good. This makes the traditional planning based approaches unsuitable for planning under strict time constraints.

Pervasive middleware has already adopting some techniques for high level planing. For example the CARISMA middleware (Capra et al., 2003), which applied a sealed-bid auction approach used in micro-economic to dynamically resolve policy conflicts during context changes. The middleware acted as auctioneer to collect bids from applications and then deliver the one with the required quality of service one where utility functions are computed during this process in order to decide who wins among the conflicting policies.

The MUSIC project [1] is applying utility functions for achieving the planing of self-adaption (Alia et al., 2007)(Rouvoy et al., 2008). The overall utility function is a weighted sum of a set of dimensional utility functions, where the weight corresponds to the importance of each QoS dimension preferred by a user. Early filtering of non-recommended or infeasible options is applied in order to improve the performance for planning. In reality, this important factor should be dynamic instead of static as very often the preferences can not be met.

The utility function used in Aura is composed of three parts: configuration preferences, supplier preferences, and QoS preferences. Then it will find the best match between user's needs and preferences for a task, and the environment's capabilities. The focus of Aura is to calculate in real time near-optimal resource allocations and re-allocations for a given task. For the service oriented computing as targeted by Hydra, service level agreement (SLA) should be considered for this kind of resource optimizations.

Software agents are beginning to be applied for the achievement of self-management capabilities, evidenced by the first international Agents for Autonomic Computing Workshop AAC 2008 [2]. Software agents based negotiation and optimization are used to achieve self-adaptation and self-optimization for power networks [3]. The application of software agents for self-diagnosis is well recognized, for example by Utton in (Utton, 2008) and also in his PhD thesis. In this work, the negotiation for fault diagnosis is based on the hybrid of FIPA Request and FIPA Propose protocols[4]. In the service oriented environments, we believe that the FIPA Contract Net and Iterative Contract Net protocol are more suitable where .

The BDI (Belief-Desire-Intention) model was invented because of the slow performance of pure planning systems which make planners less usable in dynamic environments. An exploration of applying BDI (Belief-Desire-Intention) agent for self-adaptation is discussed in (Morandini et al., 2008), which explicitly model goals, plans and their relationships. It shows

[1] http://www.ist-music.eu/
[2] http://www.iids.org/aac-2008/
[3] http://www.cs.vu.nl/ warnier/Papers/aac08.pdf
[4] http://www.fipa.org/repository/ips.php3

that adaptation requirements naturally fit for the BDI agent architecture. It also shows that high level transformation from self-adaption requirements to BDI agents implementation can be automated but the existing tool support should be extended to support model building. The goal types, faults, and recovery activities lack formal definition though, and should be integrated to the proposed framework. This work would be greatly enhanced if it is integrated with semantic web technologies to improve the goal, plan, and other activity semantics.

Genetic Algorithms (GAs) are beginning to be used for planning in self-management (Rouvoy, 2007). In MUSIC, GA is proposed for service composition and deployment, as GA is a powerful approach for multi-objective optimization. From (Rouvoy, 2007), we know that this idea is still at investigation phase, but we think GA is promising in the planning layer as GA potentially can rapidly locate good solutions, even for difficult complex search spaces. However the implementation and evaluation of the fitness function of GA is an important factor in the speed and efficiency of the algorithm.

To summarize the survey, we can see that in the service oriented computing environments like Hydra, the achievement of the Goal management layer should consider the following aspects:

- Dynamically calculating the utility of self-management activities according to dynamic contexts. These dynamic contexts are the key to trigger various corresponding self actions. This also means that the context model should reflect the dynamism needed to achieve self-management.
- Explicitly modeling the semantic of goals are necessary to unambiguously interpret them and then take actions. This can also improve the capabilities of the handling of self actions. For example, if we know the exact meaning of a self-healing goal, although the healing service may not exist locally, the self-healing can then resolve the situation with a service matching with remote self-healing services.
- The underlying system architecture should remain stable and should not dramatically changed. For example, if the underlying system is following a Repository style, it should not violate the constraints of the repository style (e.g. there should exists at least one repository component) after finishing related self-management actions. Of course it is another case if it is designed to change the architectural styles.
- Although the traditional planning algorithms have high complexity and therefore poor performance, Gat's original paper on the 3L architecture explicitly designates such algorithms as suitable for the planning layer. In the interest of maintaining good performance for the Hydra self-* managers, however, we will initially focus on approaches that promise better performance.

In the following sections, we will first formulate the self-management model, and then we will show the Hydra utility functions considering these dynamic contexts, and our algorithm used for achieving self management goals.

8.2 Hydra Self-management model

An adaptivity model is proposed in (Alia et al., 2007), which is defined as a tuple composed of components and plans. From the survey of BDI agents based planning, it is also advantageous to explicitly consider the goals in the self-management model. Considering the service orientation nature of Hydra, we propose a self-management model (*SeM*) of a

pervasive service system as:

$$SeM = (A, G, P, S, Q, C, U)$$

where A represents the set of underlying software architecture and architecture style of the pervasive systems, G stands for self-management goals set, P represents the plans set for achieving a goal, S represents the underly services set under a plan, C represents context dimensions, Q represents underlying QoS dimensions, U represents utility function for achieving a goal in the current context with a set of services in a plan, by considering QoS requirements, and the system should also meet architectural constraints after self-management activities.

Now we will elaborate the context dimensions, utility function definition, and QoS dimensions.

8.2.1 Context dimensions

We have proposed the needed context types and context models in (Zhang et al., 2007b)(Zhang and Hansen, 2008a). In summary, for user contexts, user social network, user preferences, user habit, and user mood should be considered; for resource contexts, CPU, memory, operating system, storage, libraries, and battery (power) information should be included; for environment contexts, network connections, location, time, and possibly temperature, humidity and other physical measurements for the surroundings should be considered; for security, encryption, key etc. should be considered. They can be formulated as followed:

$$User = Social + Preferences + Habit + Mood$$

$$Env = Network + Location + Time + Physical$$

$$Res = CPU + Memory + Storage + OS + Library + Battery$$

$$Sec = Encription + Key + Virtualization$$

Now we formulate the contexts C as followed:

$$C = (User, Env, Res, Sec)$$

where $User$ represents user contexts, Env stands for environment contexts, Res stands for resource contexts, and Sec represents security contexts.

8.2.2 Hydra Utility function

Utility function is promoted as the practical, easy-to-use, principled way of expressing the high level objectives in self-management (Kephart and Das, 2007)(Pyrros Bratskas, 2008). Various utility functions are proposed for different domains, for example data center resource allocation (Kephart and Das, 2007), and MADAM[5] and MUSIC utility functions for mobile applications (Alia et al., 2007)(Pyrros Bratskas, 2008). Taking into the consideration of Hydra, the utility function should consider the service orientation characteristics, and also dynamism of pervasive systems.

[5] http://www.ist-madam.org/

Assume that QoS dimensions Q are defined by a vector of *n* property dimensions as followed:

$$Q = [q_1, q_2, q_3, ..., q_n]$$

and for each QoS dimension q_i, there is an associated weight W^{q_i} to express its significance, and a service S is characterized by a set of properties P (maybe defined in meta data):

$$P = [p_1, p_2, p_3, ..., p_m]$$

These properties can be atomic for example response time, round trip time (RTT), network latency. Some of the properties are defined by these atomic properties, for example reliability.

Utility functions are in general n-dimensional functions taking as arguments values from an n-dimensional utility space (Alia et al., 2007). Therefore it is reasonable to apply a simplified approach, which defines an overall and aggregated utility as a weighted sum of the set of utility functions across contexts, user preferences, and service properties, in order to reduce the complexity and improve planning performance. This is also adopted by the MADAM project and MUSIC project.

The dimensional utility function associated to the QoS dimension q_i can be defined as:

$$F(q_i) = f(c_1, c_2, c_3, ..., c_j, ..., c_l) + f(p_1, p_2, p_3, ..., p_k, ..., p_m)$$

where $f(c_j)$ stands for the utility dependent on contexts, and similarly, $f(p_k)$ represents utility dependent on service properties.

In some situations especially those unexpected, the weighting of one or some QoS dimensions may become the most dominating ones, with the others less important. Considering the case in Section 3.3, we can see that reliability, bandwidth are the most important QoS parameters, while cost is a trivial parameter in this case. Therefore we can say that the weight of a QoS dimension depends on context.

Then the aggregated utility for a given service s in a plan p ($\forall p \in P$) for achieving a goal g ($\forall g \in G$) qualified by the QoS dimension Q is:

$$U(s) = \sum_{i=1}^{n} W^{q_i} * F(q_i), \ where \ \forall s \in S$$

This function should be maximized when choosing one of the services under consideration from a set of available services, or adaption should go ahead with the one that can maximize the utility of the whole application. At the same time, the total amount of required resources from these selected services should not exceed the available resources.

8.3 Planning within pervasive services environments in Hydra

To reduce the problem space for planning, it is important to filter out some irrelevant or low usability candidates. Here a *candidate* refers to a service or a set of services (or components who implement the service). In Hydra, currently the preparation of adaptation is at design time. Take an example of the adaptation of network connection as in scenario three, the services using wired or wireless connections are designed and ready to use when the services are in the current configuration. In addition, Hydra is adopting the service oriented architecture. Therefore, all planning work finally comes to the questions of choosing a set of services that can accomplish the current self-management work, for example self-healing, self-adaption actions. For the case of rule conflicting, we will handle it separately for different self-management purpose. For example, if there are multiple diagnosis rules triggered, then the one has the highest probability may win.

In Hydra, we propose two types of filtering: service characteristics filtering, and architectural constraints filtering.

8.3.1 Service filtering based on service characteristics

The early filtering will reduce the problem domain in order to improve the efficiency for planning. The following means of early filtering are applied:

- Service category filtering. For example, a service component for error resolving of battery drains too quickly is under the category of “LiIonBatteryDiagnosis”, under parent level of “BatteryDiagnosis” service. We can easily pin point to the needed self-diagnosis service if needed with the help of this classification.
- Service property filtering. After we get the set services of the needed category, say set Sc, Sc can be further filtered according to some static service properties, for example, required operating system, CPU, and required libraries. This filtering can also be working in a dynamic way, for example, the minimum memory requirement for running a service, can be checked against the available resources. Now we can get another services set Ss.

8.3.2 Architecture based filtering

There are two steps to achieve the architectural based filtering: filtering based on component reference relationships, and filtering based on other constraints. Here we use Limbo (our web service compiler developed in Task T4.2) as a case study, because Limbo is well documented.

- *All components are correctly referenced, no dangling components*

  Limbo is using the OSGi component model and Repository style, the rule “check_OSGi_Reference_noDetails” retrieves all Repository components in the current configuration. If a component has a reference which has cardinality of the form “1.” (at least one reference to other service), then there must be a component providing that required service. This step is actually not necessary if an OSGi DS implementation is used as it is checked by the framework.

Rule: check_OSGi_Reference_noDetails

$archstyle : CurrentConfiguration(?con) \wedge$
$archstyle : hasArchitecturePart(?con, ?comp1) \wedge$
$osgi : componentName(?comp1, ?compname1) \wedge$
$osgi : reference(?comp1, ?ref1) \wedge$
$osgi : cardinality(?ref1, ?car1) \wedge$
$swrlb : containsIgnoreCase(?car1, \text{“1.”}) \wedge$
$osgi : interface(?ref1, ?inter1) \wedge$
$osgi : interfaceName(?inter1, ?name1) \wedge$
$archstyle : hasArchitecturePart(?con, ?comp2) \wedge$
$architectureRole(?comp2, ?role2) \wedge$
$archstyle : archPartName(?role2, ?rolename) \wedge$
$swrlb : equal(?rolename, \text{“Repository”}) \wedge$
$osgi : service(?comp2, ?ser2) \wedge$
$osgi : provide(?ser2, ?inter2) \wedge$
$osgi : interfaceName(?inter2, ?name2) \wedge$
$osgi : componentName(?comp2, ?compname2) \wedge$
$swrlb : equal(?name1, ?name2)$
$\rightarrow sqwrl : selectDistinct(?comp1, ?comp2)$

Assume that we have two Repository components loaded by DS and the two components are implementing two Repository interfaces that differ only with the last operation:

```
URI getHydraOntologyExtension (File wsdlFile);
URI getHydraOntologyExtension (String wsdlFile);
```

Limbo needs to be bound to the Repository interface that has the signature as the first method. The Eclipse Equinox DS bundle binds Limbo to the first Repository component that has the lowest bundle id, without respecting the interface signature it has. If it is bound to the correct Repository, then Limbo can run successfully. But if the Repository with the lowest bundle id is the one that provides the same method but with a String parameter, Limbo will not work.

The solution to this is to provide more semantic meaning to the DS service to facilitate making choices for such situations. The OSGiComponent ontology and the Process ontology can help with fully specifying valid OSGi component references taking into interface signatures consideration. This rule is named “check_OSGi_Reference_Details” as follows.

Rule: check_OSGi_Reference_Details

$archstyle : CurrentConfiguration(?con) \wedge$
$archstyle : hasArchitecturePart(?con, ?comp1) \wedge$
$osgi : componentName(?comp1, ?compname1) \wedge$
$osgi : reference(?comp1, ?ref1) \wedge$
$osgi : cardinality(?ref1, ?car1) \wedge$
$swrlb : containsIgnoreCase(?car1, \text{“1.”}) \wedge$
$osgi : interface(?ref1, ?inter1) \wedge$
$osgi : interfaceName(?inter1, ?name1) \wedge$
$archstyle : hasArchitecturePart(?con, ?comp2) \wedge$
$architectureRole(?comp2, ?role2) \wedge$
$archstyle : archPartName(?role2, ?rolename) \wedge$
$swrlb : equal(?rolename, \text{“Repository”}) \wedge$
$osgi : service(?comp2, ?ser2) \wedge$
$osgi : provide(?ser2, ?inter2) \wedge$
$osgi : interfaceName(?inter2, ?name2) \wedge$
$osgi : componentName(?comp2, ?compname2) \wedge$
$swrlb : equal(?name1, ?name2) \wedge$
$component : componentServiceDetails(?comp1, ?pr1) \wedge$
$service : presents(?pr1, ?prservice1) \wedge$
$profile : has\_process(?prservice1, ?process1) \wedge$
$process : realizedBy(?process1, ?aprocess1) \wedge$
$process : hasInput(?aprocess1, ?input1) \wedge$
$process : parameterValue(?input1, ?ivalue1) \wedge$
$component : componentServiceDetails(?comp2, ?pr2) \wedge$
$service : presents(?pr2, ?prservice2) \wedge$
$profile : has\_process(?prservice2, ?process2) \wedge$
$process : realizedBy(?process2, ?aprocess2) \wedge$

$process : hasInput(?aprocess2, ?input2) \wedge$
$process : name(?aprocess2, ?proname2) \wedge$
$process : hasOutput(?aprocess2, ?proout2) \wedge$
$process : parameterValue(?input2, ?ivalue2) \wedge$
$process : parameterValue(?proout2, ?ovalue2) \wedge$
$swrlb : stringConcat(?str1, ?compname2, " + ") \wedge$
$swrlb : stringConcat(?str2, ?str1, ?proname2) \wedge$
$swrlb : stringConcat(?str3, ?str2, "\#") \wedge$
$swrlb : stringConcat(?str4, ?str3, ?ivalue2) \wedge$
$swrlb : stringConcat(?str5, ?str4, "\$") \wedge$
$swrlb : stringConcat(?str6, ?str5, ?ovalue2) \wedge$
$swrlb : equal(?ivalue1, ?str6)$
$\rightarrow sqwrl : selectDistinct(?ivalue1, ?comp1, ?comp2, ?str6) \wedge sqwrl : select("$*valid references*$")$

Limbo can generate pervasive service code for JME/JSE/OSGi platforms, with client and/or server as options, and also UPnP descriptions for services and devices. Some of the generation combinations are not meaningful. For example, if a UPnP component is in the component list, and the generation type is only for a web service client, then this combination is not valid as the UPnP service is only meaningful in the server. This can be specified with the following rule.

$archstyle : CurrentConfiguration(?con) \wedge$
$archstyle : hasArchitecturePart(?con, ?comp1) \wedge$
$osgi : implementation(?comp1, ?imp) \wedge$
$osgi : implementationClass(?imp, ?class) \wedge$
$swrlb : containsIgnoreCase(?class, "UPnPComponent") \wedge$
$limboGenerationType(?con, ?type) \wedge$
$swrlb : equal(?type, "Client") \wedge$
$\rightarrow sqwrl : selectDistinct(?con, ?comp1, ?imp, ?type) \wedge$
$sqwrl : select("$*invalid client generation with UPnP backend*$")$

- *Architectural style constraints are followed.*

  Limbo can generate pervasive service code for JME/JSE/OSGi platforms, with client and/or server as options, and also UPnP descriptions for services and devices. Some of the generation combinations are not meaningful. For example, if a UPnP component is in the component list, and the generation type is only for a web service client, then this combination is not valid as the UPnP service is only meaningful in the server. This can be specified with the following rule.

  $archstyle : CurrentConfiguration(?con) \wedge$
  $archstyle : hasArchitecturePart(?con, ?comp1) \wedge$
  $osgi : implementation(?comp1, ?imp) \wedge$
  $osgi : implementationClass(?imp, ?class) \wedge$
  $swrlb : containsIgnoreCase(?class, "UPnPComponent") \wedge$
  $limboGenerationType(?con, ?type) \wedge$
  $swrlb : equal(?type, "Client") \wedge$
  $\rightarrow sqwrl : selectDistinct(?con, ?comp1, ?imp, ?type) \wedge$
  $sqwrl : select("$*invalid client generation with UPnP backend*$")$

After filtering with the architectural constraints, we can get a services set Sa.

8.3.3 Applying Utility functions

We now take the scenario 3 as an example to show the usage of utility functions. Three QoS dimensions are considered, namely bandwidth, reliability, and cost. We list the value of QoS parameters as in Table 8.1. Let W^R, W^B and W^C represent the weights for reliability, bandwidth, and cost respectively, and the value of the weights are shown in Table 8.2, in the range of 1 to 10. Table 8.3 lists the utility functions for the three dimensions, defined as a set of coefficient values where each coefficient specifies the utility value for a QoS dimension.

Network	reliability	bandwidth	cost
Wired	1	1000Mbs	60$/hour
Wireless	0.96	15Mbs	15$/hour

Table 8.1: values of QoS parameters

Network	W^R	W^B	W^C
Wired	10	10	1
Wireless	3	3	8

Table 8.2: QoS dimensional weights

As we can see from Table 8.2, for the wired connection, its reliability and bandwidth are two most important factors, which are at the same time equally important, while the cost is much less important. The aggregated utility is shown in Table 8.4.

Now we show the whole planning process as in Figure 8.1. Each filtering results in a set of services, which is a subset of the set before applying the filtering. For example, after the property filtering, we get a set Sp, which is then filtered with architectural constraints, and then we can a set of Sa, as a subset of Sp.

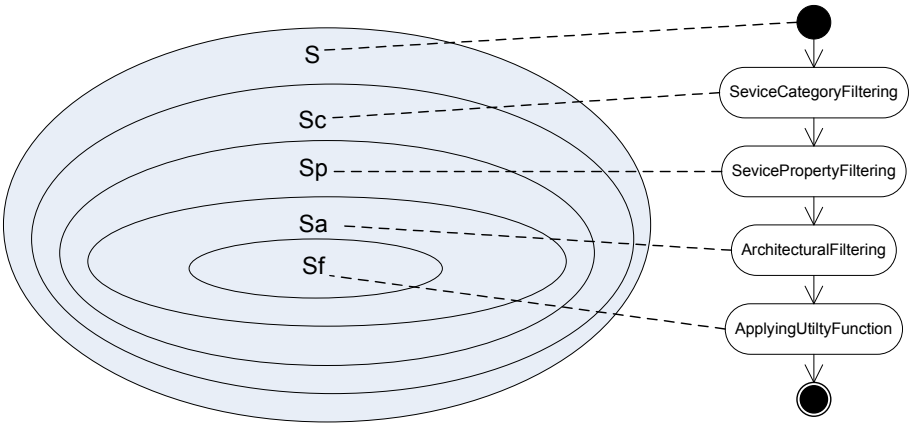

Figure 8.1: Planning process and the resulted service sets

For the final step of utility based filtering, we would also explore the application of genetic algorithms introduced in the next section.

8.4 Genetic Algorithm for planning

In the former section, we proposed the Hydra utility functions, which can synthesis multiple objectives, such as lowest cost, highest reliability and throughput in a simple way. The problem with this approach are the difficulties on the proper and precise selection of weights or utility functions to characterize a situation.

Genetic algorithms (GAs) (Mitchell, 1996) are a popular meta-heuristic that is particularly well-suited for this class of problems, which is a subclass of evolutionary algorithms (EAs).

Network	$f(R)$	$f(B)$	$f(C)$
Wired	10	9	2
Wireless	4	4	8

Table 8.3: QoS dimensional utility functions

Network	**utility**
Wired	192
Wireless	88

Table 8.4: Aggregated utility functions

It has become one of the most successful direction of computational intelligence in the past decade. GAs are well suited to solve multi-objective optimization problems, which can find a set of multiple non-dominated solutions in a single run. The ability of GA to simultaneously search different regions of a solution space makes it possible to find a diverse set of solutions for difficult problems with non-convex, discontinuous, and multi-modal solutions spaces (Konak et al., 2006). Most of the multi-objective GAs eliminate the trouble of prioritizing or weighing objectives. GA have been the most popular heuristic approach to multi-objective design and optimization problems.

8.4.1 Basic introduction of GAs

This basic introduction of GA is based on the tutorial by (Konak et al., 2006). GA are inspired by the evolutionist theory explaining the origin of species. In nature, weak and unfit species within their environment are faced with extinction by natural selection. The strong ones have greater opportunity to pass their genes to future generations via reproduction. In the long run, species carrying the correct combination in their genes become dominant in their population. Sometimes, during the slow process of evolution, random changes may occur in genes. If these changes provide additional advantages in the challenge for survival, new species evolve from the old ones. Unsuccessful changes are eliminated by natural selection.

In GA terminology, a solution is called an individual or a *chromosome*. Chromosomes are made of discrete units called *genes*. Each gene controls one or more features of the chromosome. Normally, a chromosome corresponds to a unique solution x in the solution space. This requires a mapping mechanism between the solution space and the chromosomes. This mapping is called an encoding. In fact, GAs work on the encoding of a problem, not on the problem itself.

GAs operate with a collection of chromosomes, called a *population*. The population is normally randomly initialized. As the search evolves, the population includes fitter and fitter solutions, and eventually it converges, meaning that it is dominated by a single solution.

GAs use two operators to generate new solutions from existing ones: *crossover* and *mutation*. In crossover, generally two chromosomes, called *parents*, are combined together to form new chromosomes, called *offspring*. The parents are selected among existing chromosomes in the population with preference towards fitness so that offspring is expected to inherit good genes which make the parents fitter. By iteratively applying the crossover operator, genes of good chromosomes are expected to appear more frequently in the population, eventually leading to convergence to an overall good solution.

The mutation operator introduces random changes into characteristics of chromosomes. Mutation is generally applied at the gene level. In typical GA implementations, the mutation

rate (probability of changing the properties of a gene) is very small and depends on the length of the chromosome. Therefore, the new chromosome produced by mutation will not be very different from the original one. As discussed earlier, crossover leads the population to converge by making the chromosomes in the population alike. Mutation reintroduces genetic diversity back into the population and assists the search escape from local optima.

Reproduction involves selection of chromosomes for the next generation. In the most general case, the fitness of an individual determines the probability of its survival for the next generation. There are different selection procedures in GA depending on how the fitness values are used. Proportional selection, ranking, and tournament selection are the most popular selection procedures.

The procedure of a generic GA is given as follows:

1. Set $t = 1$. Randomly generate N solutions to form the first population, $P1$. Evaluate the fitness of solutions in $P1$.
2. Crossover: Generate an offspring population Qt as follows:
   (a) Choose two solutions x and y from Pt based on the fitness values.
   (b) Using a crossover operator, generate offspring and add them to Qt.
3. Mutation: Mutate each solution $x \in Qt$ with a predefined mutation rate.
4. Fitness assignment: Evaluate and assign a fitness value to each solution $x \in Qt$ based on its objective function value and infeasibility.
5. Selection: Select N solutions from Qt based on their fitness and copy them to $Pt + 1$.
6. If the stopping criterion is satisfied, terminate the search and return to the current population, else, set $t = t + 1$ go to Step 2.

8.4.2 Algorithms for planning

There are many algorithms proposed so far as surveyed in (Konak et al., 2006), for example NSGA-II (Deb et al., 2002). For the usability in pervasive systems, efficiency is one of the most important requirements. Therefore we will choose the algorithms that shows efficiency. An interesting set of multiple objective optimization algorithms are cellular genetic algorithms, represented by its most recent algorithms called MOCell (Nebro et al., 2007) and CellDE (Durillo et al., 2008).

Cellular GAs make use of the concept of (small) neighborhood in the sense that one individual can only interact with individuals belonging to its neighborhood in the breeding loop. The overlapped small neighborhoods of cGAs help in exploring the search space: the induced slow diffusion of solutions through the population provides a kind of exploration (diversification), while exploitation (intensification) takes place inside each neighborhood by genetic operators (Durillo et al., 2008). This characteristic seems fit for the nature of pervasive systems. Also it has show that MOCell (Nebro et al., 2007) and CellDE (Durillo et al., 2008) have better performance than the existing algorithms. Therefore, they may potentially be used in Hydra.

Table 8.5 shows the comparisons of the mentioned algorithms, and compared the fitness function evaluations, diversity mechanism, whether elitism and external population is used, together with their advantages and disadvantages. The compared items are based on

Algorithm	Fitness assignment	Diversity mechanism	Elitism	External population	advantages	Disadvantages	tool support
NSGA-II	Ranking based on non-domination sorting	Crowding distance	Yes	No	Well tested, efficient	Crowding distance works in objective space only	ECJ[6], JGAP[7], JMetal[8]
MOCell	Ranking based on non-domination sorting	Cell based crowding distance	Yes	Yes	Good performance, fast convergence	Tested only with up to 3 objectives	JMetal
CellDE	Ranking based on non-domination sorting	Cell based 1-hope neighbors	Yes	Yes	Good performance, fast and elegant convergence	Tested only with up to 3 objectives	JMetal

Table 8.5: Multi-objective optimization genetic algorithms comparisons

(Konak et al., 2006), and we add the tool support section to compare whether they have implemented the compared algorithms directly (e.g. as examples) (these tools are extensible to implement any multi-objective optimization algorithms though).

We are going to explore these algorithms in the future to test the performance, and will implement at least one of them. We can predict that the algorithms can have more potential than planning in self-management. It can be used to make decisions for other Hydra components, for example the Network manager can use these algorithms to have an optimized solution on how to replicate the Hydra IDs (HIDs) considering the resources and network contexts, which have been explored by other work (Hassan et al., 2008).

9 Implementation of the semantic web based self-management in Hydra

In this section, we will first give an overall view of the Hydra run time architecture, illustrated with UML component diagram and sequence diagram. Then we will show how the semantic web based self-management (Flamenco/SW) is designed. The ASL part is detailed in Chapter 5.

9.1 Runtime view of the Hydra Self-* Architecture

Figure 9.1 shows a runtime view of the Hydra self-* architecture as a UML component diagram. Currently the ASLHost is deployed as an OSGi bundle. The dotted lines represent dependencies between components. Different layers are interacted with events, realized as publish/subscribe style, and the Hydra EventManager is used as a connector for realizing the interaction.

The bottom of the architecture is the SeMaPS ontologies/rules, in which knowledge of devices, diagnosis and error-resolution (as in Malufunction ontology) , QoS based self-management, and state based diagnosis are encoded. The Component Control layer is mainly used for state reporting (including resources state, service calling information, OSGi component adding or leaving), and updating of the related information into the corresponding self-management ontologies. Another important task is the ASLHost component, which listens for the architecture events, and then using ASL commands to bind/unbind OSGi components. The Change Management layer is used to execute rules developed based on these state and other runtime information, and parsing the inferred results in order to take actions, for example the self-healing action. For the Goal Management layer, it is used to find solutions for the malfunctions whose basic information is encoded in the Malfunction ontology, and resolve the rule conflicts based on QoS regulations or user preference etc. Utility function based evaluations of the usability of services and plans are adopted.

9.1.1 Example: Rebinding services

Figure 9.2 shows how the layers in the architecture should interact in the case of scenario number 1 given in chapter 3. The interaction among the two layers uses the EventManager for publish/subscribe. The services report (1) when they invoke another service, (2) are invoked and (3) reply to an invocation. This enables the change management layer to detect that a service, say s_2 is failing using rule *MessageCallingRelationship* based on MessageProbe ontology (by counting the number of messages exchanged from client to service). It will then discover a new service, s_3 which implements the same interface as s_1 used on s_2, and unbind s_1 from s_2 (delete its reference) and bind s_1 to s_3.

9.2 Implementation of the semantic web based self-management

The semantic web based self-management components follows the layered architecture style in essence, but also mixed with a Blackboard architecture style, and use the observer

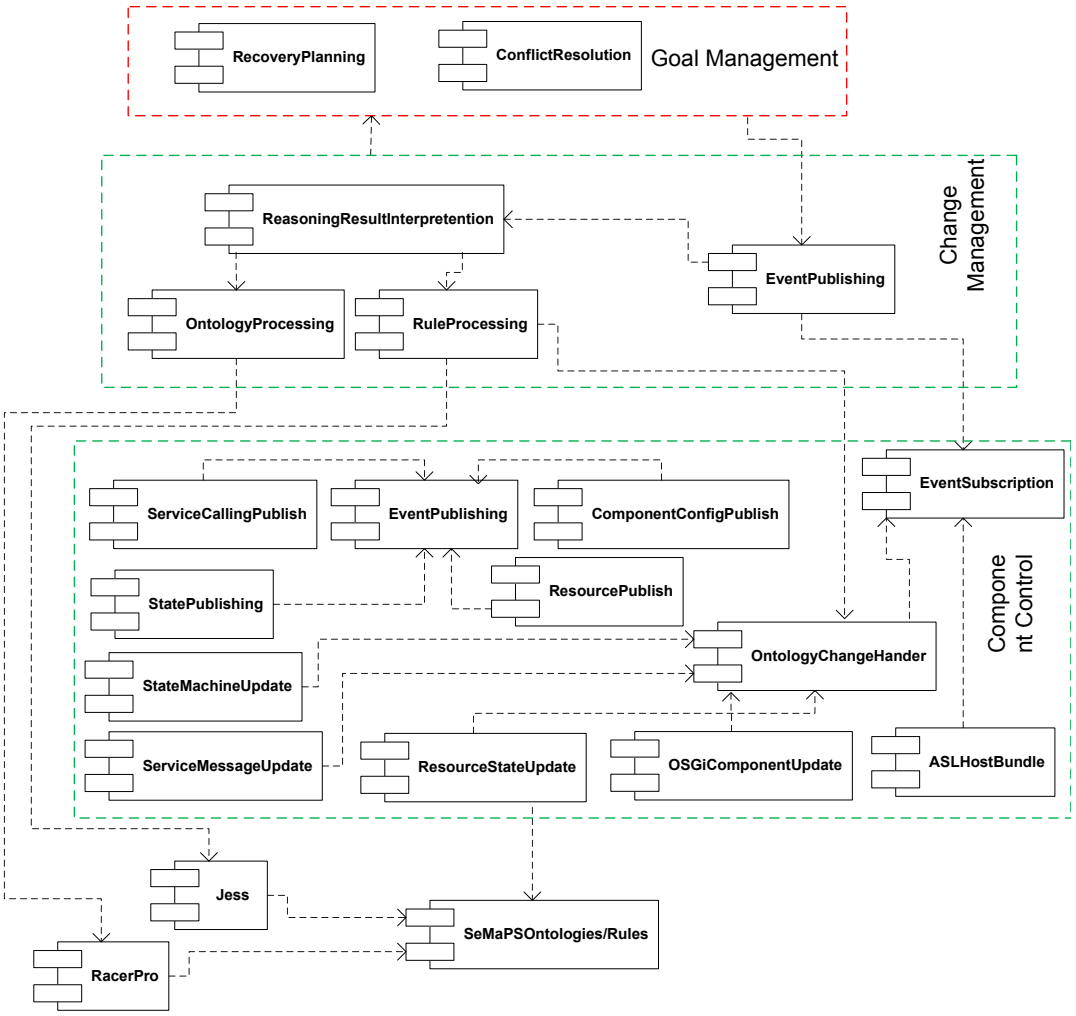

Figure 9.1: A runtime view of the overall architecture.

pattern in both the updating of state machine ontology and inferred result parsing. Assigning class responsibilities follows the GRASP patterns (Larman, 2001).

Figure 9.3 shows the high level processing sequence of the semantic web based self-managements. When there are state updates, the Flamenco/SW diagnosis component, as a subscriber to this topic, will then update the device state machine, and the updates are listened, and at the same time SWRL rule inferring will be started. New inferred information will be added to the inferred result queue, which is also observed. Next, the observer will parse the new inferred result, and publish the corresponding diagnosis result.

SWRL APIs from Protege-OWL are the only available SWRL APIs and are used for the implementation and execution of SWRL rules. The Protege-OWL code generator is used to generate Java code from the StateMachine ontology and MessageProbe ontology. The generated code provides us with convenience for allowing to use the factory class and interfaces to manipulate ontology data, such as the creation of a new state machine instance and updating instances.

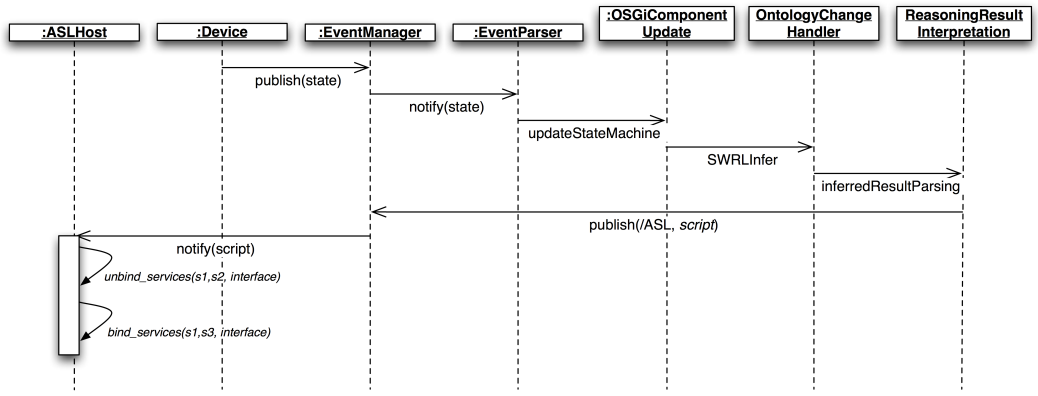

Figure 9.2: The interactions of the component control layer and the change management layer in the realization of scenario 1

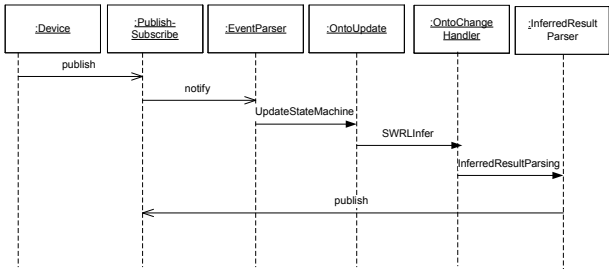

Figure 9.3: Processing sequence of the self-management components

As there is no big difference architecturally with what we present for the implementation in D4.3 (Ingstrup et al., 2008), we refer D4.3 as the source for the reader who needs further details of the implementation.

9.2.1 Brief introduction to the usage of APIs

Suppose the rules added by the knowledge developer are only related to one device. Then there is no need to do anything as the APIs can handle the diagnosis cases. In case a developer needs to process a rule, then a key class to use is RuleProcessing in package com.eu.hydra.flamenco.ruleprocessing. It can be used like this:

```
RuleProcessing rp=new RuleProcessing("http://localhost:9999/ontology/DeviceRule.owl");
HashSet<String> set=a.getAllSWRLInferred();
// get all inferred information, and can get inferred individual or property separately using
// getSWRLInferredIndividual(), getSWRLInferredProperty().
rp.checkNormalTwoColumnRule("deviceTypeChecking"); // execute rule called "deviceTypeChecking"
```

There are different methods for processing different types of rules: checkSingleColumn-Rule() which is used to process a rule returns only one column result but may have multiple rows. Similarly there are other rules processing methods.

As there may be many rules, but different rules are used for different purpose, therefore, a separate rule group can be build and executed as needed. The rule group feature can be used like this:

```
RuleGroupProcessing a=new RuleGroupProcessing("http://localhost:9999/ontology/DeviceRule.owl");
a.processRuleGroup("pig"); //create a rule group called "pig"
HashSet<String> set=a.processRuleGroup("pig"); //This will execute all rules whose name contains 'Pig'
HashSet<String> set1=a.processRuleGroup("pig", "battery", "and"); //This will execute all rules whose name contains 'Pig' and 'battery'.
HashSet<String> set2=a.processRuleGroup("pig", "battery", "or"); //This will execute all rules whose name contains 'Pig' or 'battery.
```

Now the rule grouping feature can be used to diagnosis as followed:

```
DiagnosisInitializingData.getDiagnosisInitializingDataInstance();
DiagnosisInitiation pig=DiagnosisInitiation.getPigRuleInstance();
//prepare for infered result parsing as a observer to InferredResult
InferredResultParsing parser=InferredResultParsing.getInferredResultParsingInstance();
InferredResult result=InferredResult.getInferredResultInstance();
result.addObserver(parser);
pig.Diagnosis("pig");
pig.Diagnosis("ventilator");
pig.Diagnosis("flowmeter");
```

The detailed tutorial for Flamenco is on Hydra Wiki with this link: https://hydra.fit.fraunhofer.de/confluence/display/HYDRAWIKI/WP4+Flamenco+Tutorial

10 Evaluation

10.1 Evaluating the self-diagnosis component

The self-management features are being implemented incrementally. As mentioned before, performance is the major concern for semantic web. Therefore we are still concentrating this during the evaluation, using the self-diagnosis components. We have done these evaluations in the former deliverable (Ingstrup et al., 2008), in this deliverable we have extended these evaluations by adding the rule group features, and more attempts to test the extensibility. As the development goes, we evaluate the extensibility, performance, and scalability whenever there arises the needs when new features are added.

Extensibility

Extensibility is continuously evaluated in the whole process of the self-management feature development. We started the development of the Flamenco/SW diagnosis module with the rule for temperature monitoring. Generic SWRL rule processing component is developed. Then we added the MessageProbe ontology and service calling dynamism handling to detect a service is available or not. Only piece of code was added to the reasoning result parsing component. Adding of the rule grouping feature did not affect the existing rule processing code either. Neither the adding of new ontologies affect the existing ontologies. The design has good separation of concerns nature. In summary, the rule processing and rule grouping feature is generic and has good extensibility for adding new features.

Performance

For the performance measurements, the following software platform is used: JVM 1.6.02-b06, Windows XP SP3, the hardware platform is: Thinkpad T61P T7500 2.2G CPU, 7200rpm hard disk, 2G DDR2 RAM. The time measurements are in millisecond. We adjusted heap memory to 266M for running Protege-OWL/SWRL APIs (Protege 3.4 Build 130). The size of the DeviceRule ontology is 238,824 bytes, and contains 20 rules, including 6 rules for the Smart Home system.

The performance figures are shown in Table 10.1. The *update* column represents the time needed for updating the StateMachine ontology and/or MessageProbe ontology, the *InferringTime* column shows the time needed for rules processing and inferring to obtain results, and the *AfterEventTillInferred* column shows the elapsed time, starting when the events of device state changes and/or service calling occurred, till the end of rules inferring.

Update	InferringTime	AfterEventTillInferred
843	843	843
906	906	906
922	906	922
719	719	719
953	938	938

Table 10.1: Performance before rule grouping

Instead of running all rules as a whole, we can make use of the rule grouping feature, in which a specific system, or a device, can be separately diagnosed. To achieve this, rules for the device or any other situation where a specific diagnosis is needed, can be

assigned to a rule group at runtime, and we then execute this rule group accordingly. This can greatly improve the performance as shown in Table 10.2 for the “SmartHome” rule group performance. We can see that more than 50% performance improvement is possible when rule grouping is used, with a minimum improvement of 52%, and a maximum of 69%.

Update	InferringTime	AfterEventTillInferred
328	328	328
297	281	297
297	297	297
297	281	297
344	344	344

Table 10.2: Performance after rule grouping

Scalability

As no significant changes to the whole implementation, we can notice that the scalability remains as good as before.

Discussion

The design of the rule processing and rule grouping feature is generic to be used for all self-management purposes and the overall design of the self-diagnosis has good extensibility. The testing results shown above are up to the requirements for self-management in pervasive environment, in terms of performance and scalability. The usage of the rule processing and rule grouping is via simple APIs (e.g, rule name and/or rule group name) and easy to use to develop applications.

11 Conclusions and future work

This deliverable has reported the design, implementation and initial evaluation of self-management prototypes, trying to lay a solid foundation to cover full three layer architecture, and full spectrum of self-management, including self-adaptation, self-configuration, self-healing, and self-protection.

A combined interpreter and actuator for the architectural scripting language has been implemented as an OSGi bundle for the Equinox platform, and currently enables reconfiguration in this platform; as such it provides an implementation of the 'actuator' part of the component control layer in the 3L architecture. The extension to ANT with the architectural scripting operations has likewise been designed and implemented to support the Equinox OSGi platform. It is designed to be extensible and currently constitutes a first version of the testbed.

We proposed semantic web self-management approach in Hydra motivated by the open world nature of pervasive systems. This approach is supported by a set of self-management ontologies (SeMaPS ontologies). The SeMaPS ontologies serve as the self-management knowledge base, in which both static and dynamic contexts necessary for self-management are modeled, and SWRL rules for self-adaptation, self-configuration, self-healing are implemented. SeMaPS covers also architectural ontologies which can be used to enhance self-management features by considering architectural constraints.

The self-management model and Hydra utility function is proposed to use in the Goal management after extensive survey on planning techniques for self-management planning. Genetic algorithm is investigated and promising to serve as planning algorithm due to its capability to find a global optimization candidate.

In the future, we will extend the architectural scripting features by adding atomic execution of operations, and explore the concurrent execution. At the same time, we will extend the ASL based test bed by adding more devices to it, and improve flexibility for developer to adapt the test bed.

We still envision to enhance the SeMaPS ontologies as the development goes and the usage of these ontologies provides feed back. More rules cover self-protection, self-optimization will be developed, and further to extend existing rules set. Goal management layer will be implemented based on genetic algorithm, and other algorithm will be investigated.

Bibliography

(2001). Software engineering - product quality, ISO/IEC 9126-1. Technical report, International Organization for Standardization.

Aggarwal, G., Datar, M., Mishra, N., and Motwani, R. (2004). On identifying stable ways to configure systems. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 148–153.

Ahmed, S., Ahamed, S. I., Sharmin, M., and Haque, M. M. (2007a). Self-healing for autonomic pervasive computing. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 110–111, New York, NY, USA. ACM.

Ahmed, S., Sharmin, M., and Ahamed, S. (2007b). Ets (efficient, transparent, and secured) self-healing service for pervasive computing applications. *International Journal of Network Security*, 4(3):271–281.

Alia, M., Eide, V., Paspallis, N., Eliassen, F., Hallsteinsen, S., and Papadopoulos, G. (2007). A Utility-based Adaptivity Model for Mobile Applications. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops-Volume 02*, pages 556–563. IEEE Computer Society Washington, DC, USA.

Alia, M., Horn, G., Eliassen, F., Khan, M., Fricke, R., and Reichle, R. (2006). A Component-Based Planning Framework for Adaptive Systems. *LECTURE NOTES IN COMPUTER SCIENCE*, 4276:1686.

Amundsen, S. and Eliassen, F. (2008). A resource and context model for mobile middleware. *Personal and Ubiquitous Computing*, 12(2):143–153.

Anderson, S., Hartswood, M., Procter, R., Rouncefield, M., Slack, R., Soutter, J., and Voss, A. (2003). Making autonomic computing systems accountable: The problem of human-computer. In *DEXA '03: Proceedings of the 14th International Workshop on Database and Expert Systems Applications*, Washington, DC, USA. IEEE Computer Society.

Arshad, N., Heimbigner, D., and Wolf, A. (2003). Deployment and Dynamic Reconfiguration Planning for Distributed Software Systems. In *Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*. IEEE Computer Society Washington, DC, USA.

Arshad, N., Heimbigner, D., and Wolf, A. (2007). Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality Journal*, 15(3):265–281.

Arshad, N., Heimbigner, D., and Wolf, A. L. (2004). A planning based approach to failure recovery in distributed systems. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 8–12, New York, NY, USA. ACM Press.

Babaoglu, O., Canright, G., Deutsch, A., Di Caro, G. A., Ducatelle, F., Gambardella, L. M., Ganguly, N., Jelasity, M., Montemanni, R., Montresor, A., and Urnes, T. (2006). Design patterns from biology for distributed computing. *ACM Trans. Auton. Adapt. Syst.*, 1(1):26–66.

Baresi, L., Di Nitto, E., and Ghezzi, C. (2006). Toward Open-World Software: Issues and Challenges. *COMPUTER*, pages 36–43.

Baresi, L., Di Nitto, E., Ghezzi, C., and Guinea, S. (2007). A framework for the deployment of adaptable web service compositions. *Service Oriented Computing and Applications*, 1(1):75–91.

Baresi, L. and Guinea, S. (2007). Dynamo and self-healing bpel compositions. In *Software Engineering - Companion, 2007. ICSE 2007 Companion. 29th International Conference on*, pages 69–70.

Barrett, R., Maglio, P. P., Kandogan, E., and Bailey, J. (2004). Usable autonomic computing systems: the administrator's perspective. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 18–25.

Bass, L., Clements, P., and Kazman, R. (2003a). *Software Architecture in Practice*. Addison-Wesley Professional.

Bass, L., Clements, P., and Kazman, R. (2003b). *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional.

Bhattacharya, S. and Perry, D. E. (2005). Predicting architectural styles from component specifications. In *WICSA*, pages 231–232.

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.

Chen, M., Zheng, A. X., Lloyd, J., Jordan, M. I., and Brewer, E. (2004). Failure diagnosis using decision trees. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 36–43.

Chetan, S., Ranganathan, A., and Campbell, R. (2005). Towards fault tolerant pervasive computing. *Technology and Society Magazine, IEEE*, 24(1):38–44.

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.

COSTA, P. and LASKEY, K. (2006). PR-OWL: A Framework for Probabilistic Ontologies. In *Formal Ontology in Information Systems: Proceedings of the Fourth International Conference (FOIS 2006)*. IOS Press.

Dashofy, E. M., van der Hoek, A., and Taylor, R. N. (2002). Towards architecture-based self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 21–26, New York, NY, USA. ACM Press.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *Evolutionary Computation, IEEE Transactions on*, 6(2):182–197.

Dey, A. K., Salber, D., and Abowd, G. D. (2001). A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2–4):97–166.

Diao, Y., Hellerstein, J. L., Parekh, S., Griffith, R., Kaiser, G., and Phung, D. (2005). Self-managing systems: a control theory foundation. In *Engineering of Computer-Based Systems, 2005. ECBS '05. 12th IEEE International Conference and Workshops on the*, pages 441–448.

Ding, Z., Peng, Y., and Pan, R. (2006). BayesOWL: Uncertainty Modeling in Semantic Web Ontologies. *STUDIES IN FUZZINESS AND SOFT COMPUTING*, 204:3.

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*.

Drummond, N., Rector, A., Stevens, R., Moulton, G., Horridge, M., Wang, H., and Seidenberg, J. (2006). Putting OWL in Order: Patterns for Sequences in OWL. *Proc. 2nd Workshop on OWL: Experiences and Directions*.

Durillo, J. J., Nebro, A. J., Luna, F., and Alba, E. (2008). Solving three-objective optimization problems using a new hybrid cellular genetic algorithm. In *Parallel Problem Solving from Nature - PPSN X, 10th International Conference Dortmund, Germany, September 13-17, 2008, Proceedings*, Lecture Notes in Computer Science, pages 661–670. Springer.

Eisenhauer, M., Prause, C., Schneider, A., Scholten, M., and Zimmermann, A. (2008). Updated architectural design specification. Technical Report D3.9, Hydra Consortium. IST 2005-034891.

Elkhodary, A. and Whittle, J. (2007). A survey of approaches to adaptive application security. In *SEAMS '07: Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, Washington, DC, USA. IEEE Computer Society.

Fernandes, J., Zhang, W., Ingtrup, M., Rafael, P. A., and Milagro, F. (2008). Updated architectural design specification. Technical Report D12.5, Hydra Consortium. IST 2005-034891.

Flury, T., Privat, G., and Ramparany, F. (2004). OWL-based location ontology for context-aware services. *Proc. Artificial Intelligence in Mobile Systems, Nottingham (UK)*, pages 52–58.

Ganek, A. G. and Corbi, T. A. (2003). The dawning of the autonomic computing era. *IBM Syst. J.*, 42(1):5–18.

Garlan, D., Allen, R., and Ockerbloom, J. (1995). Architectural mismatch or why it's hard to build systems out of existing parts. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 179–185, New York, NY, USA. ACM Press.

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.

Garlan, D., Monroe, R., and Wile, D. (2000). Acme: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*, pages 47–68.

Garlan, D. and Schmerl, B. (2002). Model-based adaptation for self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA. ACM Press.

Garlan, D., Siewiorek, D. P., Smailagic, A., and Steenkiste, P. (2002). Project aura: toward distraction-free pervasive computing. *Pervasive Computing, IEEE*, 1(2):22–31.

Gat, E. (1998a). *On three-layer architectures*, pages 195–210. MIT/AAAI Press, Menlo Park.

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

Goldsby, H. J., Knoester, D. B., Cheng, B. H., Mckinley, P. K., and Ofria, C. A. (2007). Digitally evolving models for dynamically adaptive systems. In *Software Engineering for Adaptive and Self-Managing Systems, 2007. ICSE Workshops SEAMS '07. International Workshop on*, page 13.

Gu, T., Punga, H. K., and Zhang, D. Q. (2005). A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications*, 28(1):1–18.

Hansen, K. M., Soares, G., and Zhang, W. (2007). Embedded Service SDK Prototype and Report. Technical Report D4.2, Hydra Consortium. IST 2005-034891.

Hansen, K. M., Zhang, W., and Fernandes, J. (2008). Osgi based and ontology-enabled generation of pervasive web services. In *15th Asia-Pacific Software Engineering Conference*, Beijing, China. To appear.

Hassan, O., Ramaswamy, L., Miller, J., Rasheed, K., and Canfield, E. (2008). Replication in Overlay Networks: A Multi-objective Optimization Approach.

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., Hansen, K. M., and Zhang, W. (2008). Self-* properties SDK prototype and report. Technical Report D4.3, Hydra Consortium. IST 2005-034891.

IST Amigo Project (2006). Amigo middleware core: Prototype implementation and documentation, deliverable 3.2. Technical report, IST-2004-004182.

Kephart, J. and Das, R. (2007). Achieving Self-Management via Utility Functions. *IEEE Internet Computing*, 11(1):40–48.

Kiciman, E. and Wang, Y.-M. (2004). Discovering correctness constraints for self-management of system configuration. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 28–35.

Konak, A., Coit, D., and Smith, A. (2006). Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering and System Safety*, 91(9):992–1007.

Kramer, J. and Magee, J. (2007a). Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, Washington, DC, USA. IEEE Computer Society.

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

Kritikos, K. and Plexousakis, D. (2007). Semantic QoS-based Web Service Discovery Algorithms. In *Web Services, 2007. ECOWS'07. Fifth European Conference on*, pages 181–190.

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.

Larman, C. (2001). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall PTR Upper Saddle River, NJ, USA.

Lightstone, S. (2007). Seven software engineering principles for autonomic computing development. *Innovations in Systems and Software Engineering*, 3(1):71–74.

Liu, H., Parashar, M., and Hariri, S. (2004). A component-based programming model for autonomic applications. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 10–17.

Maes, P. (1987). Concepts and experiments in computational reflection. *SIGPLAN Not.*, 22(12):147–155.

Mattmann, C. (2007). *Software Connectors for Highly Voluminous and Distributed Data-Intensive Systems*. PhD thesis, USC.

Medvidovic, N. and Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93.

Mehta, N., Medvidovic, N., and Phadke, S. (2000). Towards a taxonomy of software connectors. *Proceedings of the 22nd international conference on Software engineering*, pages 178–187.

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. Bradford Books.

Morandini, M., Penserini, L., and Perini, A. (2008). Towards goal-oriented development of self-adaptive systems. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 9–16. ACM New York, NY, USA.

Nami, M. R. and Bertels, K. (2007). A survey of autonomic computing systems. In *ICAS '07: Proceedings of the Third International Conference on Autonomic and Autonomous Systems*, Washington, DC, USA. IEEE Computer Society.

Nardi, D. and Brachman, R. (2003). The Description Logic Handbook-Theory, Implementation and applications.

Nebro, A. J., Durillo, J. J., Luna, F., Dorronsoro, B., and Alba, E. (2007). Mocell: A cellular genetic algorithm for multiobjective optimization. *International Journal of Intelligent Systems*, pages 25–36.

Niemelä, E., Kalaoja, J., and Lago, P. (2005). Toward an Architectural Knowledge Base for Wireless Service Engineering. *IEEE Transactions on Software Engineering*, pages 361–379.

Oreizy, P., Gorlick, M. M., Taylor, R. N., Heimhigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. S., and Wolf, A. L. (1999). An architecture-based approach to self-adaptive software. *Intelligent Systems and Their Applications, IEEE [see also IEEE Intelligent Systems]*, 14(3):54–62.

OSGi Alliance (2007). OSGi Service Platform – Service Compendium. Technical Report Release 4, Version 4.1, OSGi.

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

Pan, J., Stoilos, G., Stamou, G., Tzouvaras, V., and Horrocks, I. (2005). f-SWRL: A Fuzzy Extension of SWRL. *LECTURE NOTES IN COMPUTER SCIENCE*, 3697:829.

Parashar, M. and Hariri, S. (2005). Autonomic computing: An overview. In *Unconventional Programming Paradigms*, volume 3566 of *LNCS*, pages 257–269. Springer.

Predoiu, L. and Stuckenschmidt, H. (2008). Probabilistic extensions of semantic web languages - a survey. In *The Semantic Web for Knowledge and Data Management: Technologies and Practices*. Idea Group Inc.

Pyrros Bratskas, Nearchos Paspallis, K. K. G. A. P. (2008). applying utility functions to adaptation planning for home automation applications. In *Proceedings of the 17th International Conference on Information Systems Development (ISD2008)*. Springer Verlag.

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

Ranganathan, A., McGrath, R. E., Campbell, R. H., and Mickunas, M. D. (2003). Use of ontologies in a pervasive computing environment. *Knowl. Eng. Rev.*, 18(3):209–220.

Roman, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R. H., and Nahrstedt, K. (2002). A middleware infrastructure for active spaces. *Pervasive Computing, IEEE*, 1(4):74–83.

Rouvoy, R. (2007). Initial research results on mechanisms and planning algorithms for self-adaptation. Technical Report D1.2, MUSIC Consortium.

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.

Ruth, P., Rhee, J., Xu, D., Kennell, R., and Goasguen, S. (2006). Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure. In *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, pages 5–14.

Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., and Yan, H. (2006). Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, 32(7):454–466.

Schneider, F. B. (1984). Byzantine generals in action: implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2):145–154.

Scholten, M. and Shi, L. (2008). Quality-of-Service Enabled HYDRA Middleware. Technical Report D4.5, Hydra Consortium. IST 2005-034891.

Shaw, M. and Clements, P. (1997). A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. *Proceedings of the 21st International Computer Software and Applications Conference*, pages 6–13.

Snodgrass, R. (1988). A relational approach to monitoring complex systems. *ACM Trans. Comput. Syst.*, 6(2):157–195.

Sousa, J. P., Poladian, V., Garlan, D., Schmerl, B., and Shaw, M. (2006). Task-based adaptation for ubiquitous computing. *Systems, Man and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 36(3):328–340.

Sperandio, P., Antolin, P., and Bublitz, S. (2007). Draft of Wireless Devices Integration. Technical Report D5.4, Hydra Consortium. IST 2005-034891.

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

Srivastava, B., Bigus, J. P., and Schlosnagle, D. A. (2004). Bringing planning to autonomic applications with able. pages 154–161.

Sterritt, R., Parashar, M., Tianfield, H., and Unland, R. (2005). A concise introduction to autonomic computing. *Advanced Engineering Informatics*, 19(3):181–187.

Utton, P. (2008). An Open Diagnostic Infrastructure for Future Home Networks. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 316–320.

Walsh, W. E., Tesauro, G., Kephart, J. O., and Das, R. (2004). Utility functions in autonomic systems. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 70–77.

Weyns, D. and Holvoet, T. (2007). An architectural strategy for self-adapting systems. In *SEAMS '07: Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, Washington, DC, USA. IEEE Computer Society.

White, S. R., Hanson, J. E., Whalley, I., Chess, D. M., and Kephart, J. O. (2004). An architectural approach to autonomic computing. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 2–9.

Zhang, R., Moyle, S., Mckeever, S., and Bivens, A. (2007a). Performance problem localization in self-healing, service-oriented systems using bayesian networks. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 104–109, New York, NY, USA. ACM.

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

Zhang, W. and Hansen, K. M. (2008b). Towards Self-managed Pervasive Middleware using OWL/SWRL ontologies. In *Proceedings of the Fifth International Workshop on Modeling and Reasoning in Context (MRC 2008)*, Delft, The Netherlands.

Zhang, W., Kunz, T., and Hansen, K. M. (2007b). Product line enabled intelligent mobile middleware. In *The Twelfth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2007)*, pages 148–157, Auckland, NZ. IEEE CS.

A Published Papers

A.1 Paper 1: Semantic Web ontologies for Ambient Intelligence: Runtime Monitoring of Semantic Component Constraints

Semantic Web ontologies for Ambient Intelligence

Runtime Monitoring of Semantic Component Constraints

Klaus Marius Hansen and Weishan Zhang and Joao Fernandes and Mads Ingstrup
Department of Computer Science, University of Aarhus
Aabogade 34, 8200 Århus N, Denmark
{klaus.m.hansen,zhangws,jfmf,ingstrup}@daimi.au.dk

ABSTRACT

Semantic web-based context modeling is widely used in pervasive computing systems to achieve context awareness which is essential for Ambient Intelligence (AmI). In the Hydra middleware for pervasive services, context awareness is extended for self-management purposes, which is an integral part of Hydra. To achieve this, a set of self-management ontologies called *SeMaPS (Self-Management for Pervasive Services)* are developed, where the dynamism of device state changes and service invocation are taken into account. To show the effectiveness of these ontologies, in this paper, we focus on our Component ontology that extends OSGi's Declarative Service specification to add capabilities for expressing architectural (especially global) and functional constraints (e.g. contextual constraints), and show how to use it to verify component configurations by using a pervasive service compiler at runtime.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Validation*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*Representations (procedural and rule-based)*

General Terms

Languages, Design, Verification.

Keywords

OWL/SWRL ontology, context awareness, ambient intelligence, Configurations, OSGi declarative service

1. INTRODUCTION

Ambient Intelligence (AmI) aims to make pervasive computing[10] more usable through natural interaction, personalized and efficient services, and context awareness. Having context awareness is very important in knowing when and where a service can happen, what triggers a service provision, how to provide a service to whom and so on. Thus context awareness is essential to achieving AmI and decisions and services should be based on current or historical contexts. To enable AmI in the Hydra project (IST-2005-034891), we are building self-management capabilities into the Hydra middleware, supported by context ontologies based on semantic web technologies (OWL[1] and SWRL (Semantic Web Rule Language[2])).

Semantic web-based context modeling is arguably a powerful approach for context modeling [11], since it can provide reasoning potentials for contexts, a capability not easily achievable by other context modeling approaches. In order to support self-management, the context models should have dynamic and runtime information of the system available in order to take appropriate actions based on these dynamic contexts. Hence, the dynamic information should be reflected in the context models and used for self-management in order to make decisions on what actions should be taken to react to the changes that are needed for self awareness.

Existing pervasive computing context ontologies, such as SOUPA[1] and Amigo[4], are not targeting self-management and almost no dynamic and runtime models of the underlying pervasive systems are considered. This makes these existing ontologies unsuitable for the self-management purposes, which depend on the timely reporting of the status of devices, network, and even on running processes.

To realize our vision of semantic context awareness-based self-management in Hydra [14, 16], we designed *SeMaPS* (Self-Management for Pervasive Services), a set of self-management ontologies in Hydra. The context ontologies in SeMaPS are considering runtime contexts which are necessary for self-management. These dynamic contexts include device runtime status, service call/response relationships, and service execution time. SWRL rules are developed to handle self-management features such as malfunction diagnosis, device and system status monitoring, and service selection based on QoS parameters. When there are state changes or service calls, the dynamic running information is fed into the related self-management context ontologies, which then trigger execution of self-management rules for adaption, monitoring, diagnosis, and other aspects of self-management.

For this paper, we will demonstrate the utility of SeMaPS

[1]OWL homepage. http://www.w3.org/2004/OWL/
[2]SWRL specification homepage. http://www.w3.org/Submission/SWRL/

by focusing on one of its ontologies, Component ontology that extends OSGi's Declarative Service specification through *OWL-DS* to add capabilities for expressing architectural and functional constraints over component configurations, and show how to use it to verify the component configurations at runtime by using a pervasive service compiler (*Limbo* [3]) as an example.

The rest of the paper is structured as follows: The design of SeMaPS context ontologies and their structure are presented in Section 2. Next, we discuss how we extended OSGi DS (Section 4) followed by a more detailed case study of applying OWL-DS to the Limbo compiler (Section 5). We present the implementation of runtime validation in Section 6. Then we discuss related work in Section 7. The paper is concluded in Section 8.

2. SEMAPS ONTOLOGIES

In Hydra, to make full use of context awareness, a semantic web based self-management approach is being adopted [14, 16], with the support of the SeMaPS context ontologies. The high level structure SeMaPS ontologies is shown in Figure 1. The dynamic contexts are modeled with runtime concepts and properties in the related ontologies.

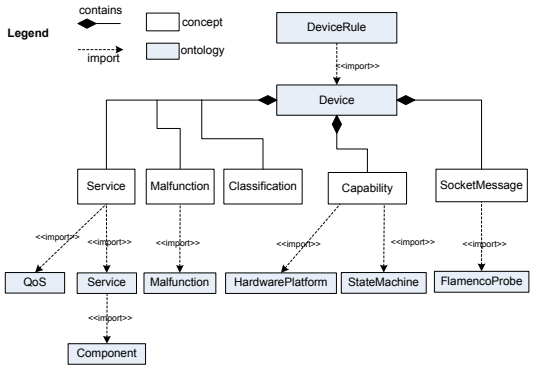

Figure 1: Structure of the SeMaPS ontologies

The Device ontology presents *HydraDevice* (as a concept) type classification (e.g. Mobile Phone, PDA, Thermometer). This is based mainly on the device classification in Amigo project ontologies [4]. To facilitate self-diagnosis, there is a concept called *HydraSystem* to model a system composed of devices that provide services. A corresponding object property *hasDevice* is added, which has the domain of *HydraSystem* and range as *HydraDevice*. The *HydraDevice* concept has a data-type property *currentMalfunction* which is used to store the inferred device malfunction diagnosis information at runtime.

The HardwarePlatform ontology defines concepts such as *CPU*, *Memory*, and relationships to devices (in the Device ontology), for example *hasCPU*. This ontology is based on the hardware description part from W3C's deliveryContext ontology[3]. Power consumption concepts and properties for different wireless network are added to the HardwarePlatform ontology to facilitate power-awareness, including a *batterLevel* property for monitoring battery consumption at runtime.

[3]Delivery Context Overview for Device Independence. `http://www.w3.org/TR/di-dco/`

The device Malfunction ontology is used to model knowledge of malfunction and recovery resolutions. It provides the classification of device malfunctions (for example, *BatteryError*). The malfunctions are defined into two categories: *Error* (including device totally down) and *Warning* (including function scale-down, and plain warning) according to severeness. There are also two other concepts, *Cause* and *Remedy*, which are used to describe the origin of a malfunction and its resolution.

The QoS ontology defines some important QoS parameters, such as availability, reliability, latency, error rate, etc. Furthermore, properties for these parameters are defined, such as their nature (dynamic, static) and impact factor. There is also a *Relationship* concept in order to model the relationships between these parameters. The QoS ontology is developed based on Amigo QoS ontology [4].

To model device state changes, a state machine ontology is developed based on [2] with many improvements to facilitate self-management work: the State concept has data-type property *isCurrent* to indicate whether a state is current or not for the purpose of device monitoring, a *doActivity* object property is added to the *State* in order to specify the corresponding activity in a state, and also a data-type property *hasResult* is added to the *Action* (including activity) concept in order to check the execution result at runtime, together with three data-type properties that are added to model historical action results in order to conduct history based self-management work.

To model the invocation of services, a FlamencoProbe ontology is developed to monitor the liveness of computing node, and facilitating the monitoring of QoS, such as the request/response time of a corresponding service call. The *SocketProcess* concept is used to model a process running in a client or service, and *SocketMessage* to model a message sent to/from between client and service. There is also a concept called *IPAddress*, which is related to *HydraDevice* with a property *hasIPAddress* in the Device ontology. The object properties *invoke*, *messageSourceIP*, and *messageTargetIP* are used to build the invoking relationships, and data type property *initiatingTime* is used to model the time stamp for a message.

The Component ontology is based on the OSGi's Declarative Service specification [5] as we are adopting OSGi as the underlying component model in Hydra. It specifies the *Component* (as a concept) dynamic status, for example whether it is *enabled*, and also static characteristics such as its *reference* to other service, its *implementation* interface, and *service*s provided. Figure 2 shows partially the details of the Component ontology.

In the following sections, we will demonstrate the capabilities of these ontologies from a different angle by applying the Component ontology and SWRL rules to verify configurations of Limbo. We will first introduce OSGi's Declarative Services (OSGi DS) on which we build the Component ontology, and then we discuss OSGi DS' shortcomings, exemplified with requirements for Limbo configurations. Then we propose an approach called *OWL-DS* based on the Component ontology and SWRL rules, to verify the configurations for Limbo at runtime. Please note that the OWL-DS approach is not limited to Limbo, but can be applied to all situations where configuration need to be verified using the OSGi component model.

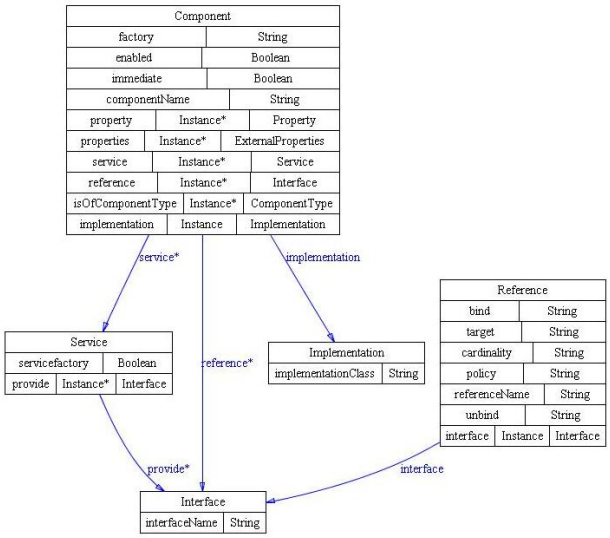

Figure 2: Component ontology (simplified)

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="com.eu.hydra.limbo">
  <implementation class="com.eu.hydra.limbo.Limbo"/>
  <service>
    <provide interface="com.eu.hydra.limbo.generator.Generator"/>
  </service>
  <reference name="BACKEND"
     interface="com.eu.hydra.limbo.backend.Backend"
     cardinality="1..n"
     policy="dynamic"
     bind="addBackend"
     unbind="removeBackend"/>
  <reference name="FRONTEND"
     interface="com.eu.hydra.limbo.frontend.Frontend"
     cardinality="1..n"
     .../>
  <reference name="REPOSITORY"
     interface="com.eu.hydra.limbo.repository"
     .../>
</component>
```

Figure 3: OSGi DS Example

3. OSGI DECLARATIVE SERVICES

OSGi provides a set of services per default [5], one of which is *Declarative Services* management. OSGi's Declarative Services Specification [5] enables developers on the OSGi platform to declaratively manage service composition at runtime. Concretely, OSGi DS allows OSGi bundle developers to provide a XML-based description of *components* that may be instantiated at runtime to provide and require services. Figure 3 shows an example of such a description which specifies the main component of the Limbo compiler.

Figure 4 shows a logical view of (a part of) Limbo's software architecture. *Limbo* provides a *Generator* service that *Frontends* and *Backends* may use. Frontends process source artifacts (such as Web Service Description Language (WSDL[4] files) whereas Backends produce output artefacts (such as web service stubs and skeletons). Both Backends and Frontends may use a single *Repository*. At runtime Limbo selects and uses a set of Backends (and Frontends) based on Limbo's

[4]http://http://www.w3.org/TR/wsdl

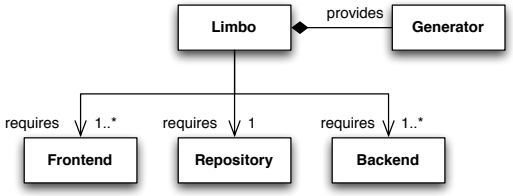

Figure 4: Limbo Logical Architecture

configuration.

This description essentially makes sure that the runtime architecture of Limbo is as shown in Figure 4: The Limbo component (inside an OSGi bundle) will be instantiated by the OSGi DS runtime and that, in this case, will need to implement the *Generator* interface since this is a service provided by the component. Furthermore, the Limbo component requires the presence of at least one *Frontend* and at least one *Backend* and one *Repository*. When these services are available, the references are said to be *satisfied* and the component may be *activated*.

Essentially, OSGi DS provides a way for components to specify provided and required services (in the form of Java interfaces) declaratively so that the OSGi framework can resolve service dependencies dynamically. Even though this is a convenient and powerful composition mechanism, we argue that it should be extended. The Limbo compiler case exemplifies a number of limitations of OSGi DS:

- *Global constraints are not supported.* This means that one cannot express architectural constraints that are non-local. An example of this could be that there must be exactly one instance of the *Repository* service for consistency reasons.
- *Contextual constraints are not supported.* The OSGi DS constraints are closed in the sense that they are specified at packaging time in the OSGi bundles. An example of where this is insufficient in the Limbo case is that it could not express that JME and OSGi server is not a legal combination.

4. EXTENDING OSGI DS THROUGH OWL-DS

We use an approach as in SAWSDL[5] to extend OSGi DS. In doing so, OSGi components reference the Component ontology and configuration rules. The Component ontology is built based on OSGi DS XML Schema as shown in Figure 2. This approach to extending OSGi DS is called OWL-DS.

To conveniently model component types, we add a concept *ComponentType* into the OWL model. This is important for specifying the constraints based on the component types. In fact, the services provided by a component (specified by interfaces) can be used to identify a component type, but this may be counter-intuitive for a user to reference the component type in this way. We are also using SWRL rules to link the *ComponentType* with the component interfaces. In order to model the configuration based on the semantic components, we use a *SystemConfiguration* concept to model the set of configurations that components can have.

[5]http://www.w3.org/TR/sawsdl/

There is a way to specify some local constraints using OWL capabilities (using cardinality restriction). OWL cardinality restrictions are referred to as local restrictions as they are stated on properties with respect to a particular class. That is, the restrictions constrain the cardinality of that property on instances of that class[6]. This is a simple and feasible way to implement constraints for component configurations. However, this approach is not sufficient to specify component configurations for our purposes for the following reasons:

Low usability : The model developer has to specify all the components as concepts (for every component) and properties explicitly, and then use the concepts and properties to specify constraints. This is not a practical way as it is hard to enumerate components in practice.

Not flexible : All constraints are explicitly stated with every component, if a new component is added, the model has to be changed accordingly. There should be a scalable way to specify constraints.

Not powerful : OWL in itself is criticized for its limited expressiveness, e.g., not being able to use math expressions or string operators that may be involved in constraints.

We are intending to use SWRL to specify the constraints for the configuration of components. This is feasible with SWRL (e.g. using the SWRL APIs from Protege[7]) because of the extensibility of SWRL.

If we know the current set of components that are available to OSGi DS, we can apply SWRL to a semantic description of this set. Here we can use SWRL built-ins, such as the mathematical built-ins, string built-ins, and Abox and Tbox built-ins. The basic idea is to retrieve these current components, and then use SWRL to specify what is a valid configuration, and what is an invalid combination following by reporting of violations of configurations.

5. LIMBO RUNTIME VALIDATION

In this section, we will use SWRL to specify the validation of component configuration. A SWRL rule is composed of an antecedent part (body), and a consequent part (head). Both the body and head consist of positive conjunctions of atoms. A SWRL rule means that if all the atoms in the antecedent (body) are true, then the consequent (head) must also be true. SWRL is built on OWL DL and shares its formal semantics. In our practice, all variables in SWRL rules bind only to known individuals in an ontology in order to develop DL-Safe rules to make them decidable. In our example SWRL rules, the symbol $\wedge$ means conjunction, and $?x$ stands for a variable, $\rightarrow$ means implication, and if there is no ? in the variable, then it is an instance.

Our approach is general and not limited to Limbo, the component model and configuration model are generic and can be applied to any cases of component semantic descriptions and component semantic configurations.

Taking Limbo as a case, the following steps are involved in the semantic validation. In practice, all the steps can be executed in a whole instead of step by step.

[6]http://www.w3.org/TR/owl-features/
[7]http://protege.stanford.edu/

Check the services required by a component. The rule retrieves all components in the current configuration. If a component has a reference which has cardinality at least one, then there must be component provide the required service. Or else, there is something wrong with the configuration. This step is not necessary if OSGi DS is used, but it is necessary if the OSGi component model applied to other situations. The rule is shown in Figure 5.

```
ComponentBased(?con) ∧
hasComponent(?con, ?comp1) ∧
osgicomponent:reference(?comp1, ?ref1) ∧
osgicomponent:cardinality(?ref1, ?car1) ∧
swrlb:containsIgnoreCase(?car1, "1.") ∧
osgicomponent:interface(?ref1, ?inter1) ∧
osgicomponent:interfaceName(?inter1, ?name1) ∧
hasComponent(?con, ?comp2) ∧
osgicomponent:service(?comp2, ?ser2) ∧
osgicomponent:provide(?ser2, ?inter2) ∧
osgicomponent:interfaceName(?inter2, ?name2) ∧
swrlb:equal(?name1, ?name2)
  → sqwrl:selectDistinct(?con, ?comp1, ?comp2, ?name1, ?name2) ∧
sqwrl:select("valid references for Configuration")
```

Figure 5: rule for checking component reference

Check component platform. All components should support the required targetted platform for Limbo compilation. A component should have this supported platform specified in its property and will be retrieved by the rule and compare with the specified targeting platform, if it is not supported, then it is not valid for this configuration. The rule is shown in Figure 6.

```
ComponentBased(?con) ∧
hasComponent(?con, ?comp1) ∧
osgicomponent:property(?comp1, ?pro1) ∧
osgicomponent:propertyName(?pro1, ?proname1) ∧
osgicomponent:value(?pro1, ?value) ∧
targetingPlatform(?con, ?plat) ∧
swrlb:notEqual(?value, ?plat)
  → sqwrl:selectDistinct(?con, ?comp1, ?pro1, ?value, ?plat) ∧
sqwrl:selectDistinct("invalid plaform combination for targeted and component supported")
```

Figure 6: rule for checking component supporting platform and the targeted platform

Check generation combination. Some of the generation combinations are not meaningful. For example, if *JME* is a targeting platform, then an *OSGi* server is not an option because OSGi is not supported on JME currently in our environment. This rule is shown in Figure 7.

Check number of component type limitations. Sometimes it is important to limit the number of component with a specific type, in a running configuration. We will first retrieve the component and assert its type according to its provided interfaces, and then count the number of this kind of components, and programatically check with its limit. This rule is shown in Figure 8.

In order to specify the limitation of different type of components a configuration has in the rule body, the SWRL builtins should be extended, and will be out of scope of

```
CurrentConfiguration(?con) ∧
currentBackend(?con, ?currentb) ∧
osgicomponent:hasComponentInstance(?currentb, ?inst) ∧
abox:hasURI(?inst, ?uri) ∧
swrlb:containsIgnoreCase(?uri, "JME") ∧
CurrentConfiguration(?con1) ∧
currentBackend(?con1, ?currentb1) ∧
osgicomponent:hasComponentInstance(?currentb1, ?inst1) ∧
abox:hasURI(?inst1, ?uri1) ∧
swrlb:containsIgnoreCase(?uri1, "OSGi")
  → sqwrl:select(?con, ?inst) ∧
sqwrl:select(?con1, ?inst1) ∧
sqwrl:select("invalid JME + OSGi server Configuration")
```

Figure 7: rule for checking OSGi server on JME platform is invalid

```
CurrentConfiguration(?con) ∧
hasComponent(?con, ?comp) ∧
osgicomponent:service(?comp, ?service) ∧
osgicomponent:provide(?service, ?interface) ∧
abox:hasURI(?interface, ?uri) ∧
swrlb:containsIgnoreCase(?uri, "Repository")
  → sqwrl:select(?comp) ∧
sqwrl:count(?comp) ∧
osgicomponent:LimboRepository(?comp) ∧
sqwrl:countDistinct(?comp)
```

Figure 8: rule for counting Limbo repository and assert a component as a repository component

open world assumption of OWL/SWRL. Therefore programatically controlling this limit is a good option for flexibility and soundness. This later approach is the one we are using.

The component reference rule (Figure 5) is a generic rule that can be applied to all other situations where the declarative service model is used. The second rule for platform checking (Figure 6) is also generic, in situations where a component's supported platform needs to be checked. The checking of components' type limits (Figure 8), can be generalized through parameterization of SWRL that will be coming later when this feature is available from SWRL APIs. The only rule that is very Limbo specific is the rule for checking the generation combination. All in all, this semantic validation algorithm can be applied to multiple situations.

6. DESIGN AND IMPLEMENTATION

We have implemented the usage of the Component ontology and Limbo runtime validation using Protege-OWL APIs, based on the ideas presented in the former sections. Specifically, we have (1) Designed and realized the Component ontology based on Declarative Services using the Protege ontology editor, (2) Extended Eclipse Equinox's[8] Declarative Services implementation to discover and maintain a model of the component instance topology, and (3) Run time monitoring and validating of Limbo components combination using Protege-OWL/SWRL APIs.

A static Component & Connector view of the implementation is shown in Figure 9

The Equinox DS bundle has been extended with a Binding Listener that knows when service are bound to (and unbound from) components. Whenever such an event happens, the Binding Listener uses the standard OSGi Event Admin that provides a topic-based publish/subscribe service. This enables the OWL-DS Monitor to maintain a model of component instances, their services, and their relationships.

[8]`http://www.eclipse.org/equinox/`

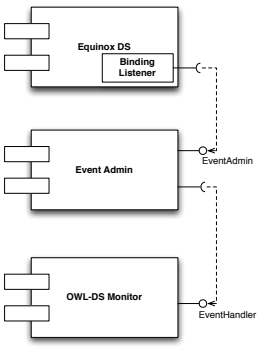

Figure 9: Static Component & Connector View of Current OWL-DS Implementation

Based on this information, the OWL-DS Monitor validates component configurations as they are made by the Equinox DS implementation, notifying whether they are valid or invalid according to semantic constraints.

7. RELATED WORK

None of the existing ontologies for pervasive computing, such as SOUPA and Amigo[4], are considering self-management concepts and requirements, however. The SeMaPS presented self-management ontologies in this paper are the key to enable various self-management tasks. At the same time, we can model complex contexts using SWRL with the Hydra ontologies [16]. Work in [7] applied SWRL-based context modeling, and illustrated three cases of applying SWRL. We go beyond it by the dynamic state-based monitoring and diagnosis using context ontologies.

Using semantic techniques (such as OWL) to extend existing component models has partially been attempted before. Sillitti and Succi[9] use XML schema to specify facets of components but they do not provide details on how to configure components. Furthermore, the implementation of the approach uses ontologies, but it is not detailed how.

In the area of product lines and feature configuration, Wang et al. [13] present a feature modelling approach in which legal feature combinations in which configuration rules are expressed with OWL. We explored this approach in the first versions of Limbo, but the approach cannot adequately specify global constraints due to the limited expressiveness of OWL. The approach of Wang et al. is furthermore mainly a design time method and is not able to cope with new contexts at runtime. The SWRL based configuration constraints that we propose could provide a more flexible solution.

An architecture description language can be used to describe configurations and styles, and then map a formally rigorous ontological specification of styles to that ADL such that its semantics are well defined. Pahl et al.[6] describe an ontology-based modeling framework for architectural styles, and show how it can be used in connection with the ACME ADL. However this work is theoretical and supports rather than overlaps with our own, in that it makes precise the benefits to an ontology based model compared to one based purely on an ADL.

Redondo et al. [8] present work to enhance the semantics of OSGi services (rather than components) to support en-

hanced service matching, based on OWL-S [9]. Redondo et al. do not use the OSGi DS specification that makes the configuration of component easier. A possible future step for us would be to incorporate OWL-S in our model to enhance service matching.

Finally, the work in [12] extends OSGi component descriptions with a special-purpose description language which is used in the CACI context awareness infrastructure. CACI proposed a context aware component model which defines that a component has application ports as well as context ports. We basically have similar mechanisms for context information such as Quality-of-Service. Additionally, we are applying semantic web technologies that have more reasoning capabilities than what is provided by [12].

8. CONCLUSIONS AND FUTURE WORK

The SeMaPS ontologies consider runtime context information of pervasive systems by incorporating pervasive service computing characteristics. These ontologies are important in supporting the envisioned semantic-web based self-management approach adopted in Hydra [14][16]. The semantic web-based self-management is suitable for the openness of pervasive computing as explored in [15]. As semantic web-based context modeling is extensively used in pervasive computing, it is beneficial to uniformly make use of these assets for self-management purposes.

Dynamic service applications require extensive development and runtime support. This paper presented OWL-DS that extends OSGi's Declarative Services with support for global (architectural) constraints on composition and support for dynamic, contextual constraints. The approach has been validated through the Limbo web service compiler case study which shows that the Limbo architecture benefits from the ability to specify advanced component constraints.

Besides self-diagnosis as presented in previous papers, we are working on extending the application of SeMaPS ontologies for quality of service-based service selection and adaption, and other self-management work which also depends on contexts. Furthermore, the investigation of applying the OWL-DS idea and the Component ontology to the .NET platform is also under investigation. This idea can be potentially used for Hydra to validate whether a configuration of an application, or the middleware is legally configured at runtime, to make sure various constraints are met dynamically.

Acknowledgments

The research reported in this paper has been supported by the Hydra EU project (IST-2005-034891).

9. REFERENCES

[1] H. CHEN, T. FININ, and A. JOSHI. An ontology for context-aware pervasive computing environments. *The Knowledge Engineering Review*, 18(03):197–207, 2004.

[2] P. Dolog. Model-driven navigation design for semantic web applications with the uml-guide. *Engineering Advanced Web Applications*, Dec. 2004.

[3] K. M. Hansen, W. Zhang, and G. Soares. Ontology-enabled generation of embeddedweb services. In *The 20th International Conference on Software Engineering and Knowledge Engineering*, pages 345–350, Redwood City, San Francisco Bay, USA, Jul. 2008.

[4] IST Amigo Project. Amigo middleware core: Prototype implementation and documentation, deliverable 3.2. Technical report, IST-2004-004182, 2006.

[5] OSGi Alliance. OSGi Service Platform – Service Compendium. Technical Report Release 4, Version 4.1, OSGi, April 2007.

[6] C. Pahl, S. Giesecke, and W. Hasselbring. An ontology-based approach for modelling architectural styles. In F. Oquendo and F. Oquendo, editors, *ECSA*, volume 4758 of *Lecture Notes in Computer Science*, pages 60–75. Springer, 2007.

[7] D.-J. Plas, M. Verheijen, H. Zwaal, and M. Hutschemaekers. Manipulating context information with swrl. *I/RS/2005/117, Freeband/A-MUSE/D3.12*, 2006.

[8] R. Redondo, A. Vilas, M. Cabrer, J. Arias, J. Duque, and A. Solla. Enhancing Residential Gateways: A Semantic OSGi Platform. *Intelligent Systems*, 23(1):32–40, 2008.

[9] A. Sillitti and G. Succi. Reuse: From components to services. In H. Mei, editor, *10th International Conference on Software Reuse (ICSR2008)*, volume 5030 of *LNCS*, pages 266–269, Beijing, China, 2008. Springer Verlag.

[10] R. Sterritt and M. Hinchey. Radical Concepts for Self-managing Ubiquitous and Pervasive Computing Environments. *LNCS*, 3825:370, 2006.

[11] T. Strang and C. Linnhoff-Popien. A Context Modeling Survey. *Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp*, pages 34–41, 2004.

[12] A. v. H. Tom Broens and M. van Sinderen. Infrastructural support for dynamic context bindings. In *1st Workshop on Semantic Web Enabled Software Engineering (SWESE'05)*, Galway, Ireland, Nov 2005. LNCS, Springer-Verlag.

[13] H. Wang, Y. F. Li, J. Sun, H. Zhang, and J. Pan. A Semantic Web Approach to Feature Modeling and Verification. In *1st Workshop on Semantic Web Enabled Software Engineering (SWESE'05)*, Galway, Ireland, Nov 2005. LNCS, Springer-Verlag.

[14] W. Zhang and K. M. Hansen. 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, Jul. 2008.

[15] W. Zhang and K. M. Hansen. Semantic web based self-management for a pervasive service middleware. In *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, Oct. 2008. To appear.

[16] W. Zhang and K. M. Hansen. 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, June 2008.

[9] `http://www.w3.org/Submission/OWL-S/`

A.2 Paper 2: Towards Self-Managed Executable Petri Nets

Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems
Venice, Italy, October 20-24, 2008

Towards Self-Managed Executable Petri Nets

Klaus Marius Hansen and Weishan Zhang and Mads Ingstrup
Department of Computer Science
University of Aarhus
{marius,zhangws,ingstrup}@daimi.au.dk

Abstract

An issue in self-managed systems is that different abstractions and programming models are used on different architectural layers, leading to systems that are harder to build and understand. To alleviate this, we introduce a self-management approach which combines high-level Petri Nets with the capability of distributed communication among nets. Organized in a three-layer Goal Management, Change Management, and Component Control architecture this allows for self-management in distributed systems. We validate the approach through the Flamenco/CPN middleware that allows for self-management of service-oriented pervasive computing systems through the runtime interpretation of Coloured Petri Nets. The current work focuses on the Change Management and Component Control layers.

1 Introduction

A self-managing, or autonomic, system is one in which technology is deployed specifically for the purpose of managing other technology [2]. Parashar and Hariri [17] describe four types of challenges for research in this area:*Conceptual* challenges concerning how we understand autonomic systems, including models and abstractions of them; the *Architectural* challenges of what architecture can enable self-management at various levels of granularity, locally or globally and so they can be specified, implemented and controlled in a predictable and robust manner; *Middleware* challenges about what core services are needed to support realization of autonomic systems subject to particular and perhaps varying quality requirements; and finally *Application* challenges that are concerned with the programming, development and maintenance of autonomic applications.

Our work addresses middleware and application challenges in the context of self-managed pervasive computing systems. Pervasive computing systems [19] are characterized by inherent dynamism, context awareness, heterogeneity, and open-endedness. All of this stresses the need for self-management and for solutions that scale across heterogeneous platforms. A central component in the Hydra middleware [9] of which this work is part, is the *Flamenco* subsystem that is responsible for self-management in a service-based pervasive computing context. This paper introduces a high-level Petri Net [10] variant, Flamenco/CPN, of this subsystem. A previous paper introduces a complementary Semantic Web-based version [22].

The rest of this paper is structured as follows: first Section 2 discusses related work from the perspective of Parashar and Hariri. Next, Section 3 presents three basic scenarios of self-monitoring and self-management that we used to design, implement, and evaluate Flamenco. The design of Flamenco/CPN is presented in Section 4, its implementation in Section 5, and its evaluation in Section 6.2. Finally, Section 7 conclude.

2 Related Work

There have been at least two main conceptual approaches to building self-managed systems. One is inspired by traditional Artificial Intelligence with the explicit representation and interpretation of plans as a basis for action, e.g. as in the three-layer architecture by Kramer and Magee [12]. The other major conceptual approach is to build systems without any explicitly represented overall plan, e.g. inspired by the decentralized control in ant colonies [13], or by evolution as an adaptation mechanism [7]. In principle our approach of communicating Petri Nets can be used for realizing both conceptual approaches, but we specifically use the Kramer and Magee model to structure self-management.

It is an open topic how well these two approaches can be unified in a given system architecture, but arguably the human nervous system that provide some of the inspiration for the vision of autonomic computing encompasses both high level cognitive explicit/conscious planning as well as relying on lower level more emergent properties for self management and healing.

From an architectural point of view, Kramer's and Magee's [12] recently proposed reference model for self-

managed systems is based on an interpretation of Gat's three layer architecture for autonomous robotics [6]. This model for self-management of software systems contrasts earlier approaches based on Sense-Plan-Act architectures [6] which have also been used in self-management systems (such as by [5]).

Kramer and Magee argue that handling self-management on an architectural level is appropriate in terms of level of abstraction and generality and casts Gat's three layer architecture in terms of conceptual layers of a self-management system:

Component Control Layer. This layer includes sensors, actuators, and simple control loops. In a self-managed system, this layer consists of elements that perform application functions ("control loops"), reporting of state to upper layers ("sensors") and facilities for creating, changing, and deleting elements ("actuators")

Change Management Layer. Based on state reported from the Component Control Layer, the Change Management Layer executes precomputed plans and change the elements in the Component Control Layer. If a conditions is met for which a plan does not exist, a new plan may be requested from the Goal Management Layer.

Goal Management Layer. This layer creates new plans based on high-level objectives of the running system. Often compute-intensive re-planning is done. Although planning has been extensively researched in AI the application of these techniques to self-management is to the best of our knowledge unexplored.

Figure 1 illustrates the reference model.

In their seminal paper on autonomic computing, Garnek and Corbi [4] describe five degrees of autonomy. In this model, increasing the level of autonomy corresponds to requiring higher-level functionality in Kramer and Magee's reference model. Since any system is unlikely to be completely autonomous there will always be certain high-level tasks a human operator must perform. Flamenco/CPN is specifically built in correspondence with the Kramer and Magee model.

From a middleware perspective we are concerned with what core services are needed to support the realization of self-managed computing systems subject to particular and perhaps varying quality constraints. Furthermore, support for distributed communication is also important here. We build on pervasive web services with embedded state machines [9] and with event-based communication using the publish/subscribe paradigm [3].

Finally, the application perspective is concerned with the programming, development and maintenance of autonomic systems. Barret et al. [1] performed ethnographic studies

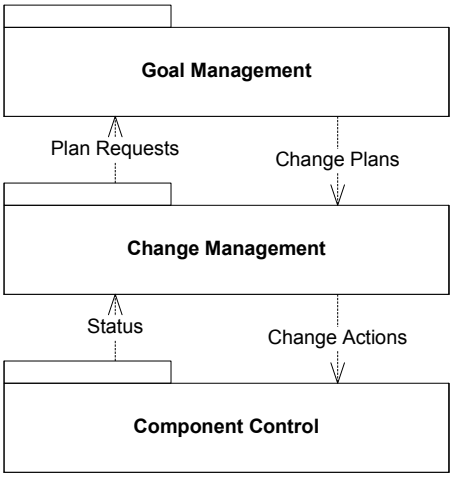

Figure 1. Three Layer Architecture Model for Self-Management

of systems administrators and emphasize the need for enabling awareness by operators, support them in rehearsal and planning activities and aid them in managing multitasking, interruptions and diversions. Similarly, we have performed ethnographic studies of agricultural, pervasive (maintenance) work [11] to derive our scenarios.

Concerning the programming of autonomic systems, [14] propose a component based framework in which each component is rule based and specified through behavior rules and interaction rules. Each component is itself self managing, a requirement also stated by White et al. [21]. We follow a similar approach in that both (device) web services and other subsystems may contain executable Petri Nets.

3 Scenarios of Self-Management

We base the description of self-management in Hydra on scenarios from the agricultural domain [11]. In the following, we outline a future scenario in which monitoring and management is involved.

> Bjarne is an agricultural worker at a large pig farm. His daily routines include taking care of one of the slaughter pig stables, maintaining equipment, and helping with various jobs on the farm as needed.
>
> To help him in his tasks, he carries around a PDA. Each morning, he feeds the pigs in one of the pig stables. When he enters the stable this morning, the PDA shows him that there is a leak in the water pipes somewhere, something that is critical for

the well-being of the pigs. He quickly locates the problem and has his smith repair the leak.

Later the same day he notices a notification that says that he needs to take care of the Heating Ventilation and Air-Conditioning (HVAC) system. Luckily there is a standby system that has been turned on, otherwise his pigs would have suffered (and possibly died) within short time.

"Time for the big service check", Bjarne mutters to himself as he sees the final notification of the day: one of the many thermometers at the farm is broken! Time to feed the pigs again...

Based on this, we derive the following scenarios of diagnosis and self-management:

3.1 Self-Management Scenario 1: Threshold

In the first part of the use scenario, Bjarne's system is able to detect a broken water pipe based on measurements of characteristics of the flow meter of the system. This is a basic scenario where diagnosis of problems is based on simple threshold values of flow meter measurements. The diagnosis is based on the fact that in a working system, the water consumption of the pigs determine the flow level of the water in the system. The thresholds are:

- *Low.* If the flow meter values are too low, the pigs are either not drinking enough water or there is too little water coming into the farm. Both of these situations are problematic.
- *High.* If the flow meter values are high (but not extremely high), it may again be a sign of problems, e.g., due to too high temperature in the stable or sickness in the pigs.
- *Too high.* In this case, if the water flow is above this level, a water pipe is most probably broken.

This example of diagnosis is taken from the farm studied where such a system is in operation. Figure 2 illustrates the case. At time *t1*, the water level is *Low*, at *t2* it is *High*, and at *t3* it is *Too high*.

3.2 Self-Management Scenario 2: Trend

In the HVAC part of the scenario, the diagnosis and repair is based on measurements from an indoor and an outdoor thermometer (see Figure 3) that is used to track the effects of the HVAC system. The diagnosis is more complicated, since a time trend of the two thermometers is needed to deduce the problem. There are two cases to consider

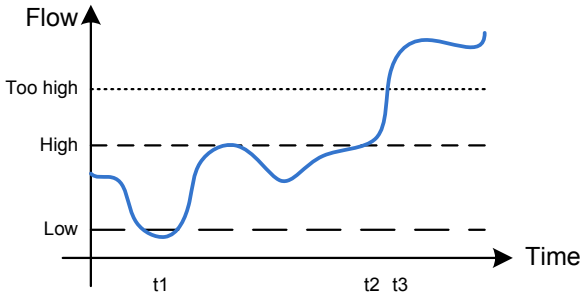

Figure 2. Self-Management Scenario 1: Threshold of Flow

1. If the temperature of either of the thermometers is below *Low* or above *High*, we cannot diagnose and assume that the system is working.

2. If the temperatures is between *Low* and *High*, we would expect the indoor temperature, *Tin*, to follow the outdoor temperature, *Tout*. If this is not the case, the spare HVAC system needs to be started.

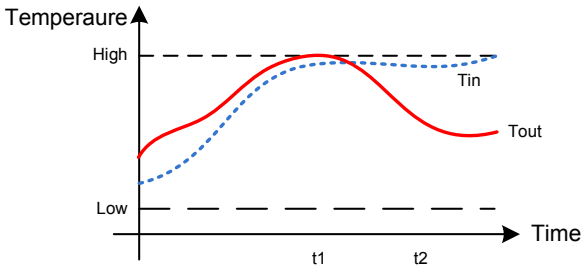

Figure 3. Self-Management Scenario 2: Trend of HVAC System

More precisely, in case 2., the situation is problematic if *Tout* falls for a number of consecutive readings whereas *Tin* does not. In the figure, *Tout* starts falling at *t1* and *Tin* should have started falling at *t2*, but it has not.

3.3 Self-Management Scenario 3: Interpretation

In the third self-management part of the use scenario, a thermometer stops working. This is detected through missing reports from the thermometer ("Status" in the Kramer and Magee model, cf. Figure 1). In this case, the user is notified about the problem which is not serious since a number of thermometers are still working. This self-management scenario illustrates the use of information from reflection on the system communication.

Concretely, this scenario assumes that Hydra is able to monitor network traffic between services and report and interpret that. Figure 4 shows the system model that the self-management of scenario 3 works upon when the network traffic has been interpreted.

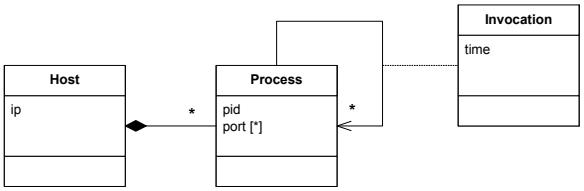

Figure 4. Self-Management Scenario 3: System Model

In this chapter, we take this as an assumption that such a model can be created and in Chapter 4, we discuss the details of how it can be created from running Hydra systems. The "Host" class models a machine on a Hydra network. The Host has an IP address and may run a number of "Processes" each of which has a process ID ("pid") [1]. Processes may communicate with other Processes (possible on other Hosts) by invocation of web services. In the model this is illustrated as an "Invocation" association class that also tracks the time of invocation. Furthermore, if a Process is a server for a web service, it will also have at least one open IP port.

In the concrete scenario, Hydra (and Flamenco) uses this information to deduce liveness of processes in the system. Figure 5 illustrates this. In the figure, the thickness of the

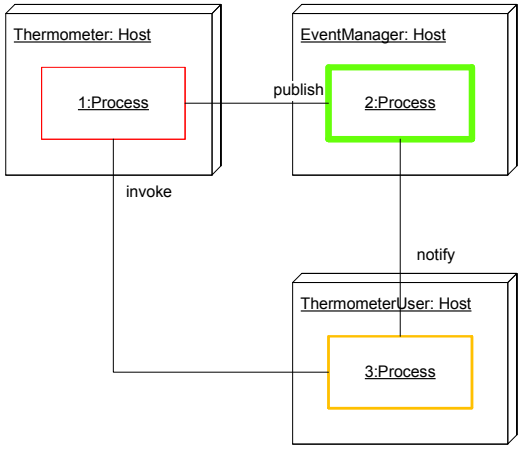

Figure 5. Self-Management Scenario 3: System Overview

[1] This is a more rudimentary model than the full Hydra network model, but it is compatible with that and will be extended

Process objects (informally) signify the perceived aliveness of the Process. Here, Process 1 which is running on the Thermometer Host is believed to be dead. This information may now be used by a Hydra application.

These three scenarios will in the following be used to illustrate the design and implementation of first iteration of the Hydra self-management approach and also to evaluate the implementation.

3.4 Further scenarios of self-management

The three scenarios outlined above are all simple, yet shows several aspects of self-management. In this section we briefly outline the full use of self-management in Hydra:

- *Goal management*. Given a goal in a specific application, a Hydra-based should continuously optimize itself to support that goal. An example from the agriculture scenario would be that the response time from an alarm is generated somewhere until it is received by a user terminal should be less than a given threshold.
- *Self-healing*. Strongly connected to goal management as outlined above is self-healing in which an application attempts to be continuously running even in the event of partial failure
- *Self-configuration*. A clear goal of much pervasive computing middleware is self-configuration in which, e.g., new devices and services announce themselves and existing services subsequently make use of these. This is supported, e.g., by UPnP. An additional layer on top of this would be that parameters of individual services should be continuously optimized based on the arrival (and departure of services)
- *Self-protection*. An example of self-protection in a self-managed system could be the ability to sense attacks and reacts upon those. Attacks could be sensed if traffic to and from the system is abnormal and a reaction to that could be to (temporarily) shut off communication to an application

4 Design of Flamenco

In this chapter, we introduce and discuss the design of Flamenco. We base the design of Flamenco on the reference model of Kramer and Magee [12]. In doing so, we are currently focusing on the Change Management and Component Control layers. These are discussed next.

4.1 Component Control

Hydra implements a service-oriented architecture based on web service interaction among devices. Thus a reason-

able granularity to build a self-management system on is the level of web service requests and responses. Furthermore, we are interested in the states of devices per se, i.e., is the device operational, stopped, not working and if it is operational what is the value of its sensor readings (if any) or its actuator state (if any).

This leads us to focus on status reporting of the following two forms:

- *State change reporting.* The Limbo web service generation tool [9] supports the generation of state machines (a restricted form of Petri Nets) describing the states of a device and its associated services. These state machines report their state changes as events through a publish/subscribe subsystem of Hydra. Furthermore, state machines may be specified in a way so that data from device services are sent as part of the events. An example of this would be a thermometer that has a *measuring* state that it continuously returns to after each physical measurement (see Figure 6)); here the event sent may naturally contain the latest temperature measurement.

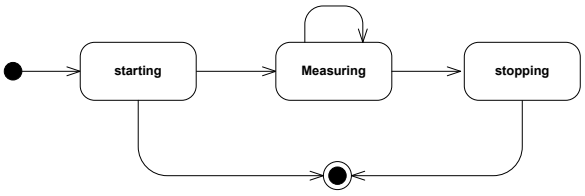

Figure 6. A thermometer state machine

- *Web service request/reply reporting.* The interaction among devices and managers in Hydra takes place via web service calls. These can then be said to correspond to "system events" in the sense of Schmerl et al. [20]. The requests and replies (and their associated data) can subsequently be used to analyze the runtime structure of running Hydra systems. To do this, we have implemented an IP sniffer that is able to report about TCP/IP packets being sent between hosts. The sniffer currently runs on Windows only. For other platforms, the Limbo compiler is able to produce adapters that report system events.

Currently, we assume that the management interface of device web services is realized through their subscription to change events, but a more refined management interface needs to be designed.

4.2 Change Management

Following up on Kramer and Magee [12], we approach self-management through *architectural abstractions*. Thus an integral part of Flamenco then becomes to make sense of system level events ("Status") and transform that into representations of, e.g., architectural components and connectors.

Schmerl et al. [20] presents a recent approach to this that uses a domain-specific language and a formal runtime semantics of this language described using *Coloured Petri Nets (CP Nets)* [10]. Flamenco/CPN generalizes this to use a state-based approach, more specifically a Petri Net-based approach, to do interpretation of system events as architectural events and to reason upon these elements.

In the following, we first briefly present CP Nets and how we use this formalism in Flamenco/CPN.

4.2.1 Coloured Petri Nets

Coloured Petri Nets is a formal, graphical modeling language with well-defined syntax and semantics. Here, we provide a very brief and somewhat informal introduction to CP-nets which is adapted from [10] and [8]. The structure of a non-hierarchical CP-net is formally defined as a tuple:

Definition 1 (Coloured Petri Net) *A* non-hierarchical CP-net *is a tuple CPN = (*Σ*,P,T,A,N,C,G,E,I), where*

- Σ *is a finite set of non-empty types called* colour sets*;*
- *P,T, and A are non-empty finite, disjoint sets of* places*,* transitions*, and* arcs*, respectively;*
- *N is a* node function *defined from A into* $(P \times T) \cup (T \times P)$*;*
- *C is a* colour function *defined from P into* Σ*;*
- *G is a* guard function *defined from T into boolean expressions;*
- *E is an* arc expression function *defined from A into expressions such that the arc expression for an arc evaluates to a multi-set of values from C(p) where p is the place that the arc is connected to; and*
- *I is an* initialization function *defined from P into expressions that do not contain variables such that the initialization expression for place p evaluates to a multi-set of values from C(p).*

4.2.2 Integrating CP-nets and Publish/Subscribe systems

An integral part of the Hydra architecture is a publish/subscribe [3] subsystem. The Limbo compiler, e.g., generates services for devices that advertise their state through notifications. A similar pattern is used in other kinds of pervasive computing middleware such as

UPnP [15]. Given our choice of CP-nets as the formalism for expressing self-management, it makes sense to consider an integration of publish/subscribe systems and CP-nets. In the following, we will thus assume that a publish/subscribe system in some way is responsible for communication to and from the Component Control layer of Kramer and Magee [12].

In defining publish/subscribe systems, we take our outset in the work of [16] and define:

Definition 2 *A* publish/subscribe system *is a tuple* $PS = (C, N, F)$*, where*

- *C is a non-empty, finite set of* client*s of the system;*
- *N is a set of* notifications*; and*
- *F is an* filter *function from N into boolean expression*

We now informally describe the runtime semantics of such systems. An execution of a publish/subscribe system is a *trace* of *states* of the system and *operations* on the system. For a formal definition, refer to [16]. The state of the system defines for each client in C which filters it has in place (what it has *subscribed* to) and which publications each client has made. The operations are on the form:

- $subscribe(c, f), c \in C, f \in F$ resulting in c subscribing to notifications that evaluate to $true$ when f is applied to them
- $unsubscribe(c, f), c \in C, f \in F$ resulting in that c does not subscribe to f
- $notify(c, n), c \in C, n \in N$ where c is notified about notification n. In a safe system, only clients that are subscribed to a filter that applied to n evaluates to true should be notified of n
- $publish(c, n), c \in C, n \in N$ meaning that c publishes a notification n. Eventually, clients, c' for which $\exists f \in F : subscribe(c', f) \wedge f(n) = true$ are notified of n

An example trace for a publish/subscribe system, ps, could be

$$\sigma = s_0, subscribe(c_1, f_1), s_1, publish(c_2, n_1), s_2, notify(c_1, n_1), s_3, ...$$

Here c_1 subscribes to f_1, c_2 publishes a notification n_1 (where $f_1(n_1) = true$) and subsequently c_1 is notified of n_1. In the trace, s_0, s_1 etc. are the states of the publish/subscribe system.

Now, to combine publish/subscribe systems and CP-nets in Flamenco/CPN, we require that certain notifications in an execution trace of a publish/subscribe system are mapped to tokens in an execution of a CP-net. More precisely, we define:

Definition 3 *A* Flamenco system *is a tuple* $(F_{PS}, F_{CPN}, p_{in}, p_{out}, F_{in}, m_{in}, c_F)$*, where* $F_{PS} = (C_{PS}, N_{PS}, F_{PS})$ *is a publish/subscribe system and* $F_{CPN} = (\Sigma_{CPN}, P_{CPN}, T_{CPN}, A_{CPN}, N_{CPN}, C_{CPN}, G_{CPN}, E_{CPN}, I_{CPN})$ *is a CP-net, and where:*

- $p_{in} \in P_{CPN}$ *is an* input place*;*
- $p_{out} \in P_{CPN}$ *is an* output place*,* $p_{out} \neq p_{in}$*;*
- $F_{in} \subseteq F_{PS}$ *is a set of* input filters*; and*
- m_{in} *is an invertible function from* N_{PS} *to* $C_{CPN}(p_{in})$
- $c_F \in C_{PS}$ *is* Flamenco client

The basic idea now is that whenever a notification is published that matches a filter in F_{in}, the notification is "converted" to a token in the executing CP-net that is placed on p_{in}. Conversely, tokens in the CP-net execution that are placed on p_{out} should be "converted" to a notification. More precisely, we want the following two properties to hold:

1. For all traces, σ, of F_{PS}, if $publish(c, n), c \in C_{PS}, n \in N_{PS} \wedge \exists f \in F_{in} : f(n) = true$ and the marking of F_{CPN} is M then there will be a subsequent marking, M', so that $m_{in}(n) \in M'(p_{in})$
2. For all markings, M, of an execution of F_{CPN}, if $\exists t \in M(p_{out})$ then the trace of F_{PS} will contain $publish(c_F, m_{in}^{-1}(t))$ and there will be a subsequent marking, M', so that $t \notin M'(p_{out})$

A simple example

Figure 7 shows a very simple example of a Flamenco CP-net. The example is cast in the context of Hydra in which the publish/subscribe system is topic-based, notifications consist of topics and events, and events are untyped (actually string-based) key-value pairs. Filtering is straight-forward in this case: clients are notified if they subscribe to the topic (or a super-topic of a topic) that another client publishes on. In Figure 7, the input place is the "Input Events" place with colour "INPUT" and the output place is the "Output Events" place with colour "OUTPUT". In the example, all publications to the topic "/statemachine/statechange" will be converted to a token on "Input Events". Whenever this happens, a token will be placed on "Output Events" and this token will eventually be converted to a notification on the topic "/tick". As a side effect, "Event Count" contains a token that counts the number of notifications that have been received on the topic "/statemachine/statechange". The count is also used in the event that is part of the "/tick" notification.

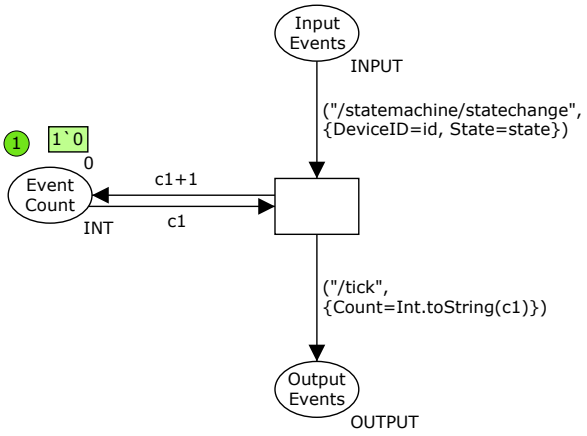

Figure 7. A Flamenco CP-net example

5 Flamenco/CPN Implementation

Flamenco/CPN is realized in a distributed fashion in the Hydra middleware. With respect to qualities, the architecture should support building a system that is adaptable and where devices are easily installable. This implies, e.g., that there should be a loose coupling between Flamenco/CPN and the rest of an application. Furthermore, efficiency is certainly an issue: there should be little impact of devices in terms of time behavior and resource utilization.

This is to a certain extent achieved by coupling Flamenco/CPN to the publish/subscribe subsystem of Hydra (i.e., the *Event Manager*) and taking advantage of that Limbo-generated services already use the Event Manager to publish state information. Further work also involves making the execution itself of the Flamenco CP-net distributed.

Figure 8 shows a runtime view of the Flamenco/CPN architecture. Here a set of Devices publish their state using

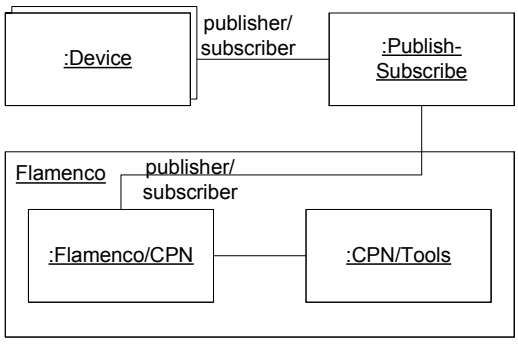

Figure 8. Flamenco/CPN architecture. Component & Connector view

the PublishSubscribe component. The Flamenco/CPN component is responsible for receiving these notifications and convert them to input CP-net tokens and for consuming output CP-net tokens when available. Flamenco/CPN does this by interacting with CPN Tools [18]. CPN Tools is responsible for the Flamenco CP-net including for the execution of it. Figure 9 shows a dynamic view of these components interacting in the form of a sequence diagram.

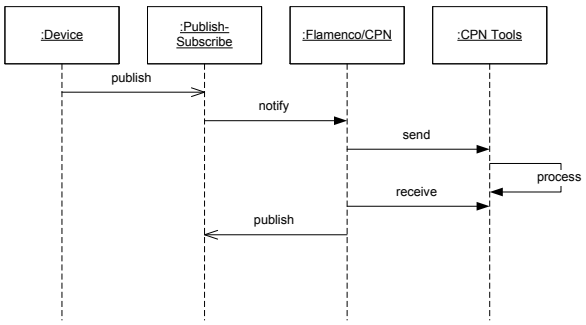

Figure 9. Flamenco/CPN architecture. Dynamic Component & Connector view

6 Evaluation of Flamenco/CPN

This section presents two types of evaluation of Flamenco/CPN:

- A qualitative evaluation in which the three scenarios from Section 3, are realized using Flamenco/CPN
- A quantitative evaluation of performance and scalability of Flamenco/CPN based on the first self-management scenario (Section 3.1)

6.1 Qualitative Evaluation

We realized the three self-management scenarios. In general, the realization of the CP-nets were straightforward in the sense that modeling was unproblematic. Work is still needed to enhance support for the programming model since CPN Tools has to be used directly and in a centralized way.

Figure 10 shows the CP-net for the first self-management scenario. The net follows a simple structure in which the three thresholds are realized as parameters (“LOW”, “HIGH”, and “TOO HIGH”). These parameters are compared with state of input from flow meters in the transitions “Low Flow”, “High Flow”, and “Too High Flow” in which the parameters and the flow meter value are used in the guards. If one of these transition fire, an alarm (on the topic “/diagnosis/alarm”) is generated.

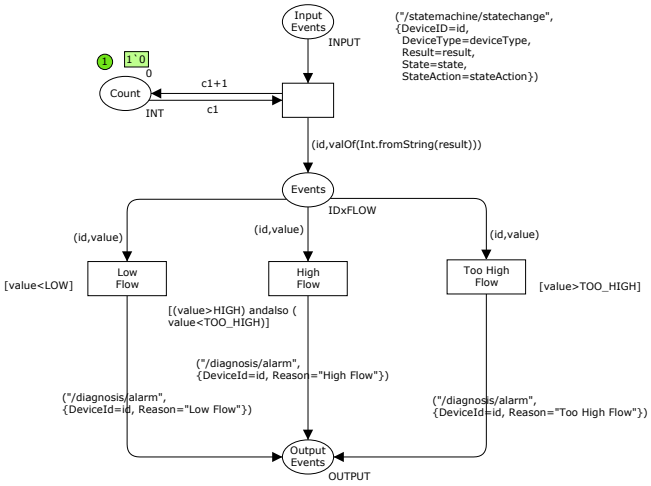

Figure 10. CP-net for scenario 1

In figure 11 we show the CP-net that implements the part of scenario 3 that pertains to constructing an architectural model of the Hydra middleware. In this case, the "Hosts" place contains a list of products of type $(host, processid)$. The "Invocations" place contains a list of invocations of type $(localhost, remotehost)$ that indicate that a process on host $localhost$ has invocation a process on host $remotehost$. The "Get Invocations" transition's guard finds invocations by assuming that two partial invocation tuples from the Flamenco sniffer belong to the same invocation (is a request and reply) if they match in hosts and ports and if the are sufficiently close in time. We did not implement the

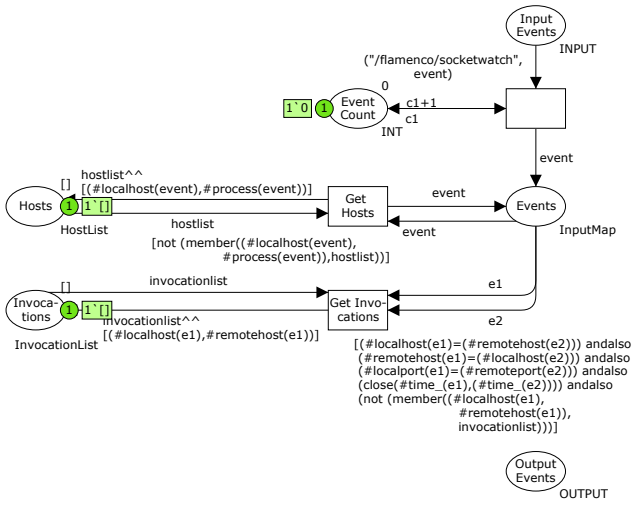

Figure 11. CP-net for scenario 3

diagnosis part of the scenario since this has been evaluated in the previous scenarios.

6.2 Quantitative Evaluation

We did a quantitative evaluation of Flamenco/CPN based on Scenario 1 from above.

The measurements were made on MacBook Pro with a 2.4 GHz Intel Core 2 Duo processor, 4 GB 667 MHz DDR2 SDRAM, and Mac OS X Version 10.5.1. Since CPN Tools only run on Linux and Windows, we ran Flamenco/CPN in VMWare Fusion Version 1.1. The tests were run only on a single machine since it was not our intention to measure network performance. Clearly, for a distributed system such as Flamenco it is of interest to measure the network-related performance, but in this case we were interested in the performance of the actual reasoning.

In the test setup, a Tester (implemented using Limbo) continually publishes state changes of a flow meter using the Event Manager. A Semaphore controls how many concurrent publications the Tester makes. If, e.g., the Semaphore has three permits, three concurrent publications can be made before the tester blocks on the Semaphore. For each publication by the Tester, Flamenco/CPN should publish an alarm event.

When a response is received from Flamenco/CPN (via the EventManager), a permit of the Semaphore is released and another publication can be made. This is used to emulate a number of devices publishing and to used to measure scalability (see Section 6.2.2). This effectively gives two parameters to vary for the evaluation:

- *Number of devices* which is controlled by number of permits of the Semaphore
- *Number of publications* which is controlled as a parameter to the Tester.

The tester publishes the total number of publications as quickly as possibly, only blocking as an effect of acquiring the Semaphore.

6.2.1 Performance

For the performance measurements, we ran the Tester with 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 500, and 1000 publication and one permit in the Semaphore. We ran each sub-scenario four times and took the average of the execution times of the last three (to allow Flamenco/CPN to start up and stabilize). The average execution time was 32.2 msec, the median was 31.0 msec and the variance was 10.2. This execution time is for a "round-trip"including execution in CPN Tools.

As Figure 12 shows, Flamenco/CPN appears to scale linearly in number of publications.

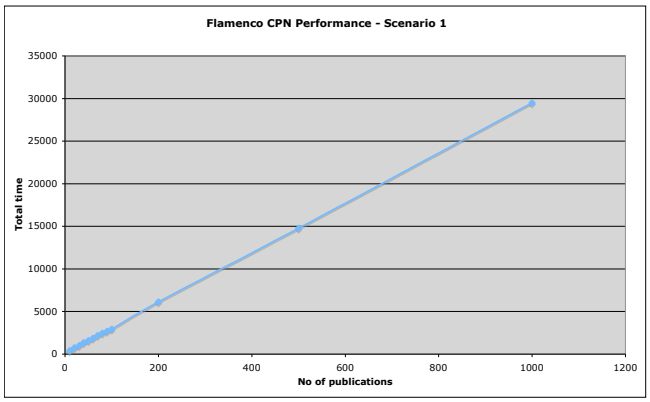

Figure 12. Flamenco/CPN performance

6.2.2 Scalability

For the scalability measurements, we simulated 1, 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100 devices each of which published 10 events. The average execution time was 29.4 msec, the median 28.5 msec, and the variance was 9.1. It can be observed that execution times are lower in average than in the performance measurements. This is most probably due to the fact that the EventManager is multithreaded and that this is exploited in the scalability tests. Furthermore, it can be noticed that the Event Manager is a bottleneck in this evaluation, preventing a real scalability evaluation. A second set of experiments where the tester created a number of concurrent threads that published state change data were created. Here Flamenco/CPN behaved well until 80 threads were running simultaneously after which Flamenco/CPN failed.

Again, cf. Figure 13, Flamenco/CPN in the test scales linearly.

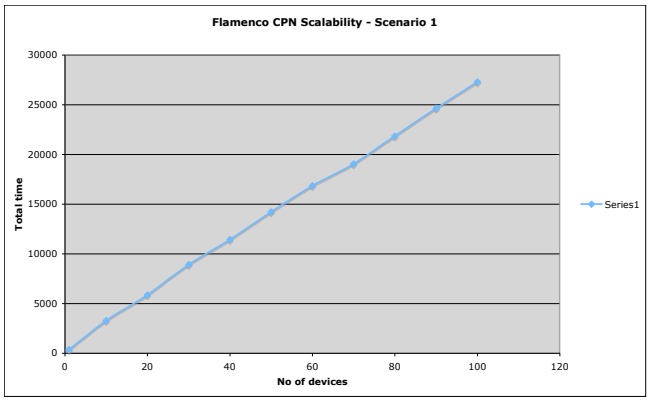

Figure 13. Flamenco/CPN scalability

7 Conclusions

This paper has introduce Flamenco/CPN, a tool for self-management in service-based pervasive computing. The tool is an instance of a general approach in which systems are modeled and implemented as communicating, high-level Petri Nets. To scale the nets to embedded devices, a restricted version of Petri Nets (essentially state machines) are used. In the reasoning part of Flamenco/CPN, full Coloured Petri Nets are used. Furthermore, Flamenco/CPN is realized according to the three-layer architecture for self-managed systems described by Kramer and Magee.

We designed and evaluated the tool in the context of agricultural scenarios, a domain with heterogeneous devices with complex interaction, exemplifying pervasive computing. Our evaluations in this context were in general encouraging: performance and scalability appears to be good and expressiveness (in terms of ability to realize the scenarios) is also good. However, further quantitative measurements are needed in particular with respect to scalability. It will, e.g., be interesting to know how the size and complexity of the Flamenco CP-nets affect performance. Moreover, memory consumption has not been measured.

Future work includes realizing a full Goal Management layer, work in which an integration with our Semantic Web-based version of Flamenco will be performed.

Acknowledgments

The research reported in this paper has been supported by the Hydra EU project (IST-2005-034891).

References

[1] R. Barrett, P. P. Maglio, E. Kandogan, and J. Bailey. Usable autonomic computing systems: the administrator's perspective. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 18–25, 2004.

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

[3] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[4] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Syst. J.*, 42(1):5–18, January 2003.

[5] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. *Proceedings of the first workshop on Self-healing systems*, pages 27–32, 2002.

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

[7] H. J. Goldsby, D. B. Knoester, B. H. Cheng, P. K. Mckinley, and C. A. Ofria. Digitally evolving models for dynamically adaptive systems. In *Software Engineering for Adaptive and Self-Managing Systems, 2007. ICSE Workshops SEAMS '07. International Workshop on*, pages 13–13, 2007.

[8] K. Hansen and L. Wells. Dynamic Design and Evaluation of Software Architecture in Critical Systems Development. In *Proceedings of the 11th Australian Conference on Safety-Related Programmable Systems*, 2006.

[9] K. M. Hansen, W. Zhang, and G. Soares. Ontology-enabled generation of embedded web services. In *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering*, Redwood City, San Francisco Bay, USA, Jul. 2008. To appear.

[10] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*. Springer, 1992.

[11] K. E. Kjaer. Ethnographic studies as a requirement gathering process for the design of context aware middleware. In *Adjunct Proceedings of Middleware 2007*, 2007.

[12] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

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

[14] H. Liu, M. Parashar, and S. Hariri. A component-based programming model for autonomic applications. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 10–17, 2004.

[15] B. Miller, T. Nixon, C. Tai, and M. Wood. Home networking with Universal Plug and Play. *Communications Magazine, IEEE*, 39(12):104–109, 2001.

[16] G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt University of Technology, Sept. 2002.

[17] M. Parashar and S. Hariri. Autonomic computing: An overview. In *Unconventional Programming Paradigms*, volume 3566 of *LNCS*, pages 257–269. Springer, 2005.

[18] A. Ratzer, L. Wells, H. Lassen, M. Laursen, and K. Jensen. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. *Applications and Theory of Petri Nets 2003: 24th International Conference, ICATPN 2003, Eindhoven, the Netherlands, June 23-27, 2003, Proceedings*, 2003.

[19] M. Satyanarayanan et al. Pervasive computing: vision and challenges. *IEEE Personal Communications*, 8(4):10–17, 2001.

[20] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering architectures from running systems. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 32(7):454–466, 2006.

[21] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, and J. O. Kephart. An architectural approach to autonomic computing. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 2–9, 2004.

[22] W. Zhang and K. M. Hansen. Towards Self-managed Pervasive Middleware using OWL/SWRL ontologies. In *Proceedings of the Fifth International Workshop on Modeling and Reasoning in Context (MRC 2008)*, Delft, The Netherlands, Jun. 2008.

A.3 Paper 3: Semantic Web based Self-management for a Pervasive Service Middleware

Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Venice, Italy, October 20-24, 2008

Semantic Web based Self-management for a Pervasive Service Middleware

Weishan Zhang and Klaus Marius Hansen
Department of Computer Science, University of Aarhus
Aabogade 34, 8200 Århus N, Denmark
{zhangws,klaus.m.hansen}@daimi.au.dk

Abstract

Self-management is one of the challenges for realizing Ambient Intelligence in pervasive computing. In this paper, we propose and present a semantic web based self-management approach for a pervasive service middleware where dynamic context information is encoded in a set of self-management context ontologies. The proposed approach is justified from the characteristics of pervasive computing and the open world assumption and reasoning potentials of semantic web and its rule language. To enable real-time self-management, application level and network level state reporting is employed in our approach. State changes are triggering execution of self-management rules for adaption, monitoring, diagnosis, and so on. Evaluations of self-diagnosis in terms of extensibility, performance, and scalability show that the semantic web based self-management approach is effective to achieve the self-diagnosis goals, and lay a solid foundation for further self-management work.

1 Introduction and Motivation

Several aspects characterize a pervasive computing system [16] compared to the traditional computing system, among them the prominent ones are: *Openness and Dynamism*: pervasive systems are often open in the sense that any device or service can come and go any time and anywhere. *Sharing*: knowledge on pervasive services should be shared to different service consumers in order to make the service provision conducted in a quality-of-service-aware way, and of course this process should be done in a secured way. *Context awareness*[6]: it is important to know when and where the service can happen, what triggers a service provision and how to provide a service to whom. And more recently, *self-management*: which is the enabler towards dependable pervasive system that leads to higher quality of pervasive systems. The self-management [13] includes a broad list of features, such as self-configuration, self-adaptation, self-optimization, self-protection and self-healing (through self-diagnosis), which are important for achieving dependability for pervasive systems towards the vision of Ambient Intelligence (AmI).

In fact, these different characteristics are inter-related. In our vision, context awareness is the key feature for enabling the others. For example, self-protection can be achieved by taking into consideration of the current context a user is in and then choose an appropriate security mechanism to protect information. A service can be shared based on the location context and quality of service (QoS) requirements, and at the same time QoS can also be considered as a part of context. The semantic web based context modeling provided by OWL (Web Ontology Language)[1] ontologies, will help to make the openness and dynamism more manageable due to the Open World Assumption (OWA) [11] adopted by OWL/SWRL(Semantic Web Rule Language)[2].

The semantic web based context modeling is promoted as a powerful way for context modeling [17], which can provide reasoning potentials for what contexts we are in, a capability not easily achievable by other context modeling approaches. This is vital to achieve the vision of self-management that should come with a pervasive service middleware.

In this paper, we present a semantic web based self-management approach in Hydra [3], supported by a set of self-management ontologies. The context OWL ontologies are considering run time contexts, such as device run time status and service call/response relationships. SWRL rules are developed to handle self-management features, such as malfunction diagnosis, device and system status monitoring, and service selection based on QoS parameters. When there are state changes or service calls, the dynamic run time information is fed into the related self-management context ontologies using an eventing mechanism. The state changes are triggering the execution of self-management SWRL rules for adaption, monitoring, diagnosis, and so

[1] OWL homepage. http://www.w3.org/2004/OWL/.
[2] SWRL specification homepage. http://www.w3.org/Submission/SWRL/
[3] http://www.hydra.eu.com

on. Evaluations of self-diagnosis in terms of extensibility, performance, and scalability show that the semantic web based self-management approach are effective to achieve self-diagnosis goals, and lay a solid foundation for further self-management work.

The rest of the paper is structured as follows: Section 2 presents the mechanisms used for run time state reporting necessary for self-management. Then in Section 3 we present a hybrid approach for context modeling in Hydra. The design of self-management context ontologies and their structure are shown in Section 4. In Section 5 we discuss the rationale of semantic web usage for self-management in pervasive computing. Section 6 presents the architecture and design of self-management based on OWL/SWRL ontologies of the Hydra middleware; Section 7 demonstrates the proposed approach using the Hydra Diagnosis Manager together with some evaluations. We compare our work with the related work in Section 8. Conclusions and future work end the paper.

2 State reporting for self-management

The Hydra project is developing self-managed middleware for pervasive embedded and network systems based on service-oriented architecture. One of the key issue for implementing self-management in pervasive computing is to get the timely information of the run time status of all devices, service calls and network connections. The Hydra middleware is based on web services, therefore two types of state reporting via an event publish/subscribe system are utilized in Hydra:

- *Application-layer*[4] *reporting* in which application-specific state is reported to self-management components.
- *Network-layer reporting* in which general data about communication is reported to self-management components.

2.1 Application-Layer Reporting

In many self-management scenarios in Hydra it is important to be able to reason on and to change the (application-specific) state that a device or a service is in. Embedded devices are often implemented as one or more *state machines* in which events cause state changes of the device or its services and may cause effects. We thus assume that Hydra-enabled devices have an embedded state machine that may be used to provide application-layer reporting at runtime. The generation of such state machines are supported by the Limbo compiler [9]. In the following we precisely formulate the integration of such state machines.

The general idea of state reporting on the application layer is that actions, activities, and transition occurrences may result in state reporting. To report application-specific data via the state machine, we allow the state machine designer to make use of the services that a device offers. Figure 1 shows a conceptual diagram of the connection between devices, services, and statemachines. The relationship between Service and Port is akin to the concepts described in the WSDL 1.1 specification[5]. The state machines

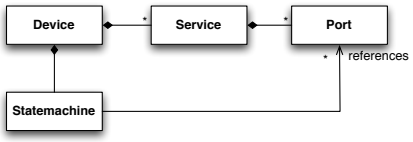

Figure 1. Conceptual View of Devices and Services

can reference its associated Ports by using the port name. Furthermore, events are published for transitions such as in:

send notification({Result=Port.service()})

The result of this is to create a notification of either type transition or activity (cf. Table 1) and add an attribute with name "Result" and value being the result on invoking "service" on "Port".

Figure 2 shows a (very simplified) state machine for an example thermometer device. The thermometer device has a service with a port named "TH03" that has an operation called "getTemperature". In the example, the thermometer implementation will continuously measure the temperature, calculate a temperature and send a "measured" event when a new temperature has been calculated. The effect of the

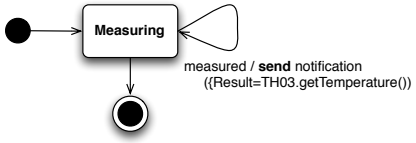

Figure 2. Simplified state machine for a Thermometer

transition occurring is the publication of a notification, an example of which is shown in Table 1.

[4]Here we use the terms "application layer" and "network layer" in the sense of the OSI reference model [20]

[5]http://www.w3.org/TR/wsdl

Topic	
/statemachine/transition	
Attribute	**Value**
DeviceId	pico_th03_0xA12A
DeviceType	PicoTh03
FromStateName	Measuring
ToStateName	Measuring
EventName	measured
Result	17.54

Table 1. An Example Transition Notification

2.2 Network-Layer Reporting

The network reporting is used to deduce whether processes have failed/stopped responding or if performance of network communication is adequate, and also potentially to calculate the service responding time. As Hydra is using SOAP[6] for web service calls, therefore our interests for protocols are HTTP, SOAP, TCP, UDP and IP [20]. In order to get report from network, a number of approaches are feasible:

- *Instrument the service implementation*, e.g., by instrumenting (web) services themselves to log information.
- *Instrument the service middleware*, e.g., by installing a data collector in the application servers running the (web) services.
- *Instrument the service host*, e.g. by packet sniffing on the host running the (web) services, and
- *Instrument the service environment*, e.g., by passing all communication through a proxy server for logging information

The first approach is what actually happens on the application layer (cf. Section 2.1). Both the first and the second approach are only concerned with the application layer on top of HTTP, i.e., SOAP since they intercept traffic in the application. A less intrusive approach which we are using is to instrument the service host, e.g, by means of a packet sniffer. Several tools are available for this such as tcpdump[7] and Wireshark[8]. This approach has, e.g., been used by [1] to do "blackbox" debugging of distributed systems. Our current implementation uses a Windows-specific version of tcpdump (windump).

2.3 Instrumenting the calling client and the service Host

Figure 3 shows the intended deployment of our instrumenter for service hosts, Flamenco Probe. It is intended to be deployed on all hosts from which network events should be reported. Currently, we are instrumenting both of the

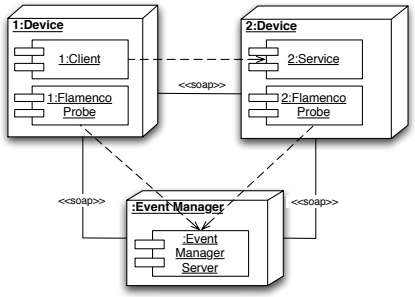

Figure 3. Flamenco Probe Deployment

client and service, in which events will be reported when a client call a service, when the service begins to serve, when the service finishes its service, and when the client get response from the service. The corresponding published events have content *ClientStart*, *ServiceStart*, *ServiceEnd*, and *ClientEnd* respectively. These events are then notified to the self-management component (Flamenco). Figure 4 shows a dynamic view of the interaction between the client, service, Flamenco Probe and the Event Manager. The left part of the figure shows the distribution where the four events occurs.

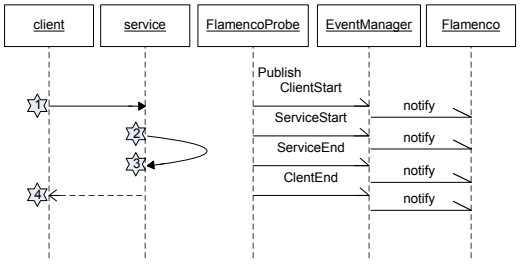

Figure 4. Flamenco Probe Dynamic View

3 A hybrid approach for context modeling in Hydra

The definition of *context* in [6] is general enough to cover different kind of contexts in pervasive computing. When self-management is concerned, it should be noted that not only static knowledge, but also dynamic and runtime context should be considered in order to handle runtime requirements. For example, if there is a malfunction, we can run a status check of a system at runtime, and monitor the dynamic contexts of the system and then make decisions on where the problem is, why the problem happens, and how

[6] http://www.w3.org/TR/soap/
[7] http://www.tcpdump.org/
[8] http://www.wireshark.org/

to tackle the problem. Context-awareness, especially the awareness of dynamic context information, is the most important factor to fulfill the goal of various self-management processes.

Various context models are compared in [17], in terms of:

- *distributed composition (dc)*: pervasive systems are intrinsically distributed, therefore the context model should be consistent with this nature.
- *partial validation (pv)*: Partially validating contextual knowledge because of the distributed composition.
- *richness and quality of information (qua)*: Capable to express rich set of contexts in pervasive systems.
- *incompleteness and ambiguity (inc)*: Capability to express incomplete and/or ambiguous information.
- *level of formality (for)*: Describing contextual facts and interrelationships in a precise and traceable manner.
- *applicability to existing environments (app)*: Possibility to apply to existing infrastructure.

Besides these aspects, it is necessary to add more when the characteristics of pervasive computing are considered in order to get a complete view of the approach(es) to be used. The following can be added:

- *Reasoning capability (rca)*: The openness of the pervasive system implies that there are something unexpected may happen anytime, where the underlying context model should be able to cope with this in order to decide the appropriate context a user is in and hence provide appropriate service to fulfill the need.
- *Resource efficiency (re)*: resource efficiency (re): To facilitate resource-constrained devices to make use of the context information, the context model should also be resource efficient.
- *developer usability (du)*: As the developer should make use of the context model to develop applications, context models should be "user-friendly" to the developer.

Take into the existing evaluation in [17], we summarize our evaluation of the context models in Table 2:

As can be seen from Table 2, an OWL ontology model is the best approach which provides good reasoning potentials for realization the vision of AmI in Hydra. On the other hand, the key-value pair model is very simple and resource-efficient for small devices. Therefore a hybrid approach for context modeling can be applied in Hydra. On the application level for a powerful node, an OWL model is used

Approach/criteria	dc	pv	qua	inc	for	app	re	du	rca
Key-value	-	-	–	–	–	+	++	+	–
Markup	+	++	-	-	+	++	-	+	–
Graphic	–	-	+	-	+	+	+	+	–
Object orientation	++	+	+	+	+	+	-	+	–
Logic based	++	-	-	-	++	–	–	-	+
Ontology based	++	++	+	+	++	+	–	-	++

Table 2. Context model comparisons

and should be used if intelligence reasoning is important. Key/value pairs are used for performance if resource efficiency is important. Limited resource node will by default use key/value pairs, and will delegate to its proxy if it needs more intelligent solutions as key/value pairs are hardly powerful enough to model complex structure contexts. This hybrid context modeling approach can be achieved by a normalization component in the context manager, in which it can target different context models such as the key/value pairs or OWL ontology approach. The overall architecture of the hybrid context modeling and the usage of context for self-management is shown in Figure 5.

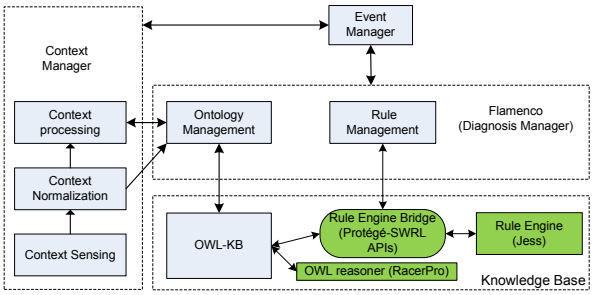

Figure 5. Architecture of a hybrid context modeling and semantic based self-management

The state reporting that we discussed in Section 2 is an example of using key/value pair as the context modeling approach. The event topic serves as key and the content of an event is the corresponding context value. Then this context event is notified to the self-management component to conduct self-management work for example self-diagnosis as shown later.

As discussed in [19], OWL itself is not powerful enough to express the complex context, for example GPS distance calculation. Also OWL itself is not capable of expressing certain additional constraints such as the rules for QoS parameter selection. In this case, we are applying SWRL to achieve the extra power for specifying self-management rules [18].

The overall idea for this semantic web based approach is: A set of self-management context ontologies are used to model the dynamic status of a underlying pervasive system,

according to the state reporting mechanisms introduced in Section 2. State changes are reported using the Hydra Event Manager, and these changed states are fed into related context ontologies. The state changes are then triggering the execution of SWRL rules which are used to monitor, configure, adapt and diagnose the underlying system.

4 Hydra ontologies for self-management

4.1 Architecture of self-management ontologies

Semantic web ontologies are widely used in pervasive computing for achieving context-awareness, but none of the existing pervasive computing ontologies are considering self-management related concepts as required by Hydra. The openness and dynamism of pervasive computing, and the nature for pervasive and embedded devices running as state machines, motivate the development of Hydra self-management context ontologies, whose high level structure is shown in Figure 6. This includes a Device ontology, Malfunction ontology, StateMachine ontology, FlamencoProbe ontology, and QoS ontology.

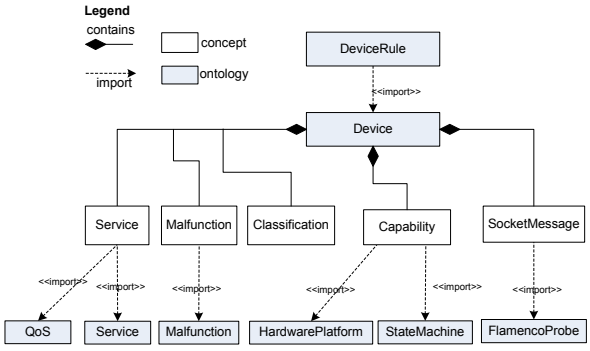

Figure 6. Structure of the Hydra context ontologies for self-management

The Device ontology is a high level ontology importing low level self-management ontologies. It is used to define basic information of a Hydra device, for example device type classification (e.g. mobile phone, PDA, sensor), device model and manufacturer, and so on, where the device type classification is based mainly on the Amigo project ontologies [12]. Some concepts and properties are specially defined for facilitating self-management. For examples, the *HydraDevice* concept has a data-type property *currentMalfunction* which is used to store the inferred device malfunction diagnosis information at run time. There is a concept called *HydraSystem* to model a system composed of devices to provide services. A corresponding object property *hasDevice* which has the domain of *HydraSystem* and range as *HydraDevice*.

The HardwarePlatform ontology is used to describe the device resources. It is based on the hardware description part from W3C's deliveryContext ontology[9]. The HardwarePlatform ontology defines major resources concept, such as CPU, Memory, Network connection capabilities, and also relationships between them, for example "hasCPU". To facilitate energy-awareness, power supply information for example battery and wired power are also modeled in this ontology. Power consumption concepts and properties for different wireless network are added to the HardwarePlatform ontology, including a *batterLevel* property for monitoring battery consumption.

The device Malfunction ontology is used to model knowledge of malfunction and recovery resolutions. It is the key ontology for self-diagnosis, which defines malfunctions categories: *Error* (including device totally down) and *Warning* (including function scale-down, and plain warning), and their sub-categories, for example, BatteryError. There are also two other concepts, *Cause* and *Remedy*, which are used to describe the origin of a malfunction and its resolution.

The QoS ontology defines some important QoS parameters, such as availability, reliability, latency, error rate, etc. And also properties for these parameters, such as its nature (dynamic, static) and the impact factor. There is also a *Relationship* concept in order to model the relationships between these parameters. The QoS ontology is developed based on Amigo QoS ontology [12].

4.2 Dynamic context modeling for self-management

According to Section 2, the dynamic context information should reflect run time status of the underlying system, and can be used to make decisions for diagnosis and monitoring, service selection based on QoS, and so on. As introduced above, some dynamic contexts are modeled with runtime concepts and properties in the related ontologies, for example the Malfunction ontology, QoS ontology, and other concepts and properties in the Device ontology, such as *currentMalfunction* and *HydraSystem*. The *currentMalfunction* will be used to store the current diagnosis information for the malfunction case, *HydraSystem* is used to dynamically model devices joining and leaving and reflect the composition of a system.

There are also two other dedicated ontologies for the achievement of self-management, namely a StateMachine ontology and a FlamencoProbe ontology.

As a common practice, mobile and embedded devices used in pervasive environments are usually designed and operated as state machines. Therefore we could make use

[9]Delivery Context Overview for Device Independence. http://www.w3.org/TR/di-dco/

of the state information to achieve self-management goals. In line with this idea, a state machine ontology is developed based on [7] with many improvements facilitating the self-management work. For example, to know the service execution history in order to know whether the service is normal, a data-type property *hasResult* is added to the *Action* (including activity) concept in order to check the execution result at runtime, at the same time three data-type properties are added to model historical action results.

Based on the state reporting mechanism introduced in Section 2, a FlamencoProbe ontology is developed to monitor the liveness of computing node, and facilitating the monitoring of QoS, such as the request/response time of a corresponding service call. The FlamencoProbe ontology has concept *SocketProcess* for modeling a process running in a client or service, and *SocketMessage* to model a message sent to/from between client and service. There is also a concept called *IPAddress*, which is related to HydraDevice with a property *hasIPAddress* in the Device ontology. There are important object properties such as *invoke*, *messageSourceIP*, and *messageTargetIP*, and data type properties for example *initiatingTime* to model the time stamp for a message.

5 Rationale for applying OWL/SWRL to self-management in Pervasive computing

The Open World Assumption asserts that knowledge of a system is incomplete, which means that if a statement cannot be inferred from what is expressed in the system, it still cannot be inferred to be false. In the OWA, statements about knowledge that are not included in or inferred from the knowledge explicitly recorded in the system may be considered unknown, rather than wrong or false in the closed world assumption. The OWA applies to the knowledge representation where the system can never be known to have been completely described in advance, which is quite consistent with the characteristics of pervasive computing systems, which are intrinsically open and dynamic. Therefore it is natural to apply the open world assumption to the pervasive computing system where the characteristics of OWA can be utilized to build the concept of open world software [4].

5.1 benefits of applying OWL/SWRL to pervasive computing

OWL and SWRL are adopting the open world assumption. They can be used to achieve the needed features of pervasive computing, such as the context-awareness and knowledge reuse across all systems. From the AmI point of view, the general benefits of applying the OWL/SWRL for the pervasive computing can be summarized as followed.

- *Deriving new information not existing in model explicitly*: This includes the deriving of new relationships between concepts and properties.

  For example, JavaVM is defined as something that can run JavaByteCode, and SuperWaba is necessarily to run JavaByteCode, therefore SuperWaba is a subclass of JavaVM. This is helpful for someone who is not familiar with SuperWaba. The same kind of reasoning will classify LeJOS [10] as a kind of Java virtual machine. This capability is valuable for the developer to understand the large variety of hardware/software platforms for pervasive systems.

  $JavaVM \equiv \exists runs.JavaByteCode, SuperWaba \sqsubseteq Library$
  $SuperWaba \sqsubseteq \exists runs.JavaByteCode$
  Then $SuperWaba \sqsubseteq JavaVM$

  The usage of transitive property can also derive useful new information, for example *requiresMoreMemory* is transitive, we have the following axioms: $requiresMoreMemory(CDC\ CLDC)$,
  $requiresMoreMemory(J2SE\ CDC)$
  Then $requiresMoreMemory(J2SE\ CLDC)$
  This is used to derive that Java SE requires more memory than CLDC. Here CDC, CLDC and J2SE are instances of its corresponding classes. Another example is the often used location context example: if John is in Room 17, Room 17 is in the Hopper Building, then John is in the Hopper Building. All other entailment such as *subClassOf*, *subPropertyOf*, *disjointWith*, and *inverseOf* can provide such capabilities.

- *Deriving additional information complementing existing knowledge*: An example of applying SWRL for deriving additional information (which means that if the battery level of a mobile phone is less than 10%, then it is a device that has very low battery) is like the following:

  $$MobilePhone(?device) \wedge hasHardware(?device, ?hardware) \wedge primaryBattery(?hardware, ?battery) \wedge batteryLevel(?battery, ?level) \wedge swrlb : lessThanOrEqual(?level, 0.1) \rightarrow VeryLowBattery(?device)$$

An SWRL rule as above is composed of an antecedent part (body), and a consequent part (head). Both the body and head consist of positive conjunctions of atoms. An SWRL rule means that if all the atoms in the antecedent (body) are true, then the consequent (head) must also be true. SWRL is built on OWL DL and shares its formal semantics. In our practice, all variables in SWRL rules bind only to known individuals in an ontology in order to develop DL-Safe rules that are decidable. In our example SWRL

[10] LeJOS homepage. http://lejos.sourceforge.NET/

rules, the symbol "$\wedge$" means conjunction, and "$?x$" stands for a variable, "$\rightarrow$" means implication, and if there is no "?" in the variable, then it is an instance.

5.2 SWRL based on self-management example

Besides the capabilities of checking whether the self-management knowledge model is valid, minimally redundant, and consistent, the OWL/SWRL based model has nice features that are needed for self-management in pervasive computing environments with the capabilities of building rules for selecting security resolution, service selection based on QoS, and so on. We now show an example of the network monitoring of service execution, based on the FlamencoProbe ontology.

In accordance with the network layer state reporting, we developed a complex rule which can calculate the round trip calling for a service, service execution time, and build the invocation relationships between processes, as shown in Figure 7. This rule first retrieves all the messages that are supposed to be a complete round trip call from a client to the service, then calculates the related information using SWRL built-in functions.

The query results from this rule can be used further to monitor the liveness of a service, and also the condition of the corresponding network connection. Similarly the rules for selection of service, security strategy can be developed. All these rules are encoded in the DeviceRule ontology as shown in Figure 6, taking into consideration of the current performance of Protege-OWL/SWRL APIs [11].

```
ipsniffer:messageID(?message1, ?message1id) ∧
ipsniffer:messageID(?message2, ?message2id) ∧
ipsniffer:messageID(?message3, ?message3id) ∧
ipsniffer:messageID(?message4, ?message4id) ∧
swrlb:equal(?message1id, ?message2id) ∧
swrlb:equal(?message2id, ?message3id) ∧
swrlb:equal(?message3id, ?message4id) ∧
abox:hasURI(?message1, ?u1) ∧
abox:hasURI(?message2, ?u2) ∧
abox:hasURI(?message3, ?u3) ∧
abox:hasURI(?message4, ?u4) ∧
swrlb:containsIgnoreCase(?u1, "clientbegin") ∧
swrlb:containsIgnoreCase(?u2, "servicebegin") ∧
swrlb:containsIgnoreCase(?u3, "serviceend") ∧
swrlb:containsIgnoreCase(?u4, "clientend") ∧
ipsniffer:messageSourceIP(?message1, ?ip1) ∧
ipsniffer:ipaddr(?ip1, ?ipa1) ∧
ipsniffer:ipaddr(?ip2, ?ipa2) ∧
ipsniffer:hasMessage(?process1, ?message1) ∧
ipsniffer:hasProcessID(?process1, ?pid1) ∧
ipsniffer:messageTargetIP(?message1, ?ip2) ∧
ipsniffer:initiatingTime(?message1, ?time1) ∧
ipsniffer:messageSourceIP(?message2, ?ip3) ∧
ipsniffer:messageTargetIP(?message2, ?ip4) ∧
ipsniffer:ipaddr(?ip3, ?ipa3) ∧
ipsniffer:ipaddr(?ip4, ?ipa4) ∧
ipsniffer:messageTargetPort(?message2, ?port2) ∧
ipsniffer:hasMessage(?process2, ?message2) ∧
ipsniffer:hasProcessID(?process2, ?pid2) ∧
ipsniffer:initiatingTime(?message2, ?time2) ∧
ipsniffer:messageSourceIP(?message3, ?ip5) ∧
ipsniffer:messageTargetIP(?message3, ?ip6) ∧
ipsniffer:ipaddr(?ip5, ?ipa5) ∧
ipsniffer:ipaddr(?ip6, ?ipa6) ∧
ipsniffer:messageTargetPort(?message3, ?port3) ∧
ipsniffer:hasMessage(?process3, ?message3) ∧
ipsniffer:hasProcessID(?process3, ?pid3) ∧
ipsniffer:initiatingTime(?message3, ?time3) ∧
ipsniffer:messageSourceIP(?message4, ?ip7) ∧
ipsniffer:messageTargetIP(?message4, ?ip8) ∧
ipsniffer:ipaddr(?ip7, ?ipa7) ∧
ipsniffer:ipaddr(?ip8, ?ipa8) ∧
ipsniffer:messageTargetPort(?message4, ?port4) ∧
ipsniffer:hasMessage(?process4, ?message4) ∧
ipsniffer:hasProcessID(?process4, ?pid4) ∧
ipsniffer:initiatingTime(?message4, ?time4) ∧
temporal:duration(?d1, ?time1, ?time4, temporal:Milliseconds) ∧
temporal:duration(?d2, ?time2, ?time3, temporal:Milliseconds)
  →ipsniffer:invoke (?message1, ?message2) ∧
sqwrl:select(?ip1, ?ipa1, ?pid1, ?ipa2, ?port2, ?pid2, ?d1, ?d2)
```

Figure 7. FlamencoProbe calling rule

6 Architecture of self-management in Hydra

Several components are involved in achieving the self-management features, based on the self-management context ontologies where dynamic contexts are encoded. These components include a Diagnosis Manager (called *Flamenco)*, which is used to monitor the system conditions and states in order to fulfill error detection, diagnosis, and provide recovery solutions; and a QoS Manager negotiating QoS parameters with other services and manages resources accordingly. Further, the QoS Manager provides device specific information to the Diagnosis Manager, and coordinates with Service Manager, Ontology Manager and Orchestration Manager. Context events are managed using an Event Manager where publish/subscribe functionality is provided.

The architecture for the self-management components are following the three Layered architecture proposed by Kramer and Magee [14] as shown in Figure 8, in which the *Goal Management*, *Component Control* and *Change Management* are enclosed with dashed line. The bottom of the architecture is the ontologies/rules, in which knowledge of devices, rule based QoS, and state based diagnosis are encoded. The Component Control layer is mainly used for state reporting (including reporting from FlamencoProbe, resources including battery level monitoring from a Resource manager). Another very important task for the Component Control layer is the updating of the related information into the corresponding self-management ontologies in correspondence to these reported events. The Change Management layer is used to execute rules developed based on these state and other run time information, and parsing the inferred results in order to take actions for example the self-healing action. For the Goal Management layer, it is used to find solutions for the malfunctions whose basic information is encoded in Malfunction ontology, and resolve the rule conflicts based on QoS regulations or user preference etc.

When there are state changes, the corresponding events are published, the device state machine instance in the StateMachine ontology will be updated. When there are web service calls, the Flamenco Probe events are published and the corresponding call information is fed into the Fla-

[11] http://protege.stanford.edu/

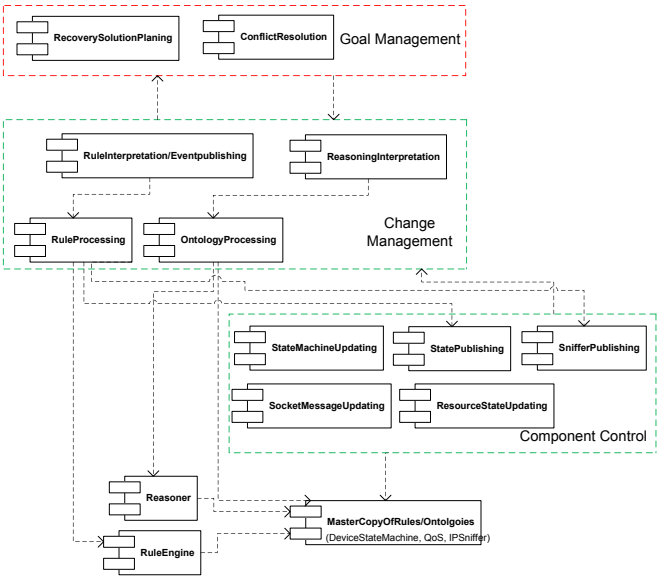

Figure 8. Self-management architecture in Hydra

mencoProbe ontology. Other run time information for resources (e.g, battery level) is from the Resource manager and is updated into the Hardware ontology. The Diagnosis Manager and QoS Manager are event subscribers to the state machine state change events and Flamenco Probe events, and these events trigger the diagnosis of the device status, executing the SWRL rules to monitor the health status of devices, monitoring QoS and also triggering the reasoning of possible device errors and their resolutions in the Goal Management layer.

7 Evaluating semantic web based self-management with Diagnosis Manager

The self-management feature of Hydra is implemented incrementally. In this iteration, we are focusing on the Diagnosis Manager, and hence at this stage we will evaluate the semantic web based self-management approach with the Diagnosis Manager. Because of potential performance problems of the semantic web based approach, the evaluation are mainly targeting performance, leaving accuracy for diagnosis and rule conflict resolution as the future work.

We started the development of Diagnosis Manager with the rule for temperature monitoring for a "Pig" farm in the agriculture domain. After finishing the implementation and testing, we then try to handle a flow meter diagnosis rules. We only need to add the flow meter rules to the existing rules set. No single line of Diagnosis Manager code needs to be changed. Then the network reporting features are added to Flamenco. The adding of this quite new feature needs the FlamencoProbe ontology, and its corresponding Java classes (generated using Protege-OWL java code generator), and then a class for handling the update of the FlamencoProbe ontology is developed. Also as expected, an extra event subject called "FlamencoProbe/socketwatch" is subscribed. All other rule processing code remains the same. After that, the Diagnosis manager is integrated with other Hydra components for the diagnosis of a weather station. We only need to develop the weather station rules, and it functions very well without the need to change any existing code. In summary, the Diagnosis Manager has good extensibility.

For the measurement of performance, the following software platform is used: Protege 3.4 Build 130, JVM 1.6.02-b06, Heap memory is 266M, Windows XP SP3. The hardware platform is: Thinkpad T61P T7500 2.2G CPU, 7200rpm hard disk, 2G DDR2 RAM. The time measurements are in millisecond. The size of DeviceRule ontology is 238,824 bytes, and contains 20 rules, including 6 rules for the Pig system, 12 generic rules which can be used in a number of domains, 3 rules (2 are shared with Pig rules) for the Weather Station, and 1 rule for FlamencoProbe related rules which is the biggest rule in the DeviceRule ontology.

The performance figures are shown in Table 3. The *update* column represents the time needed for updating the StateMachine ontology and/or FlamencoProbe ontology, the *InferringTime* column shows the time needed for rules processing and inferring to get results, and the *AfterEventTillInferred* column shows the time needed starting when the events of device state changes and/or service calling are notified, till the end of rules inferring.

When compared to performance figures in [19], we can see here the performance is worse. This has several reasons: first is that we have a larger DeviceRule ontology (238,824 bytes vs 210,394 bytes); second is that the difference of OS (vista vs. XP); third is that this version of Protege-owl has worse performance than its former release.

Update	InferringTime	AfterEventTillInferred
843	843	843
906	906	906
922	906	922
719	719	719
953	938	938

Table 3. Performance before rule grouping

Instead of running all rules in a whole, we have implemented the rule grouping features, in which a specific system, or a device can be separately diagnosed. To achieve this, rules for the device or any other situation where a specific diagnosis is needed, can be defined into a rule group at run time, and then execute this rule group accordingly. This can greatly improve the performance. Table 4 shows the

"Pig" rule group performance. We can see that more than 50% performance improvement when rule group is used, with a minimum improvement of 52%, and a maximum of 69%.

Update	InferringTime	AfterEventTillInferred
328	328	328
297	281	297
297	297	297
297	281	297
344	344	344

Table 4. Performance after rule grouping

We also did measurements on scalability which shows that time taken is in linear with the events needed to be processed, which is consistent with our former tests [19].

In summary, the testing results shown above are up to the requirements for self-management in pervasive environment as targeted by Hydra, in terms of extensibility, performance, and scalability. The majority of code base developed for the Diagnosis Manager will be used for other self-management features, especially the part for SWRL rules processing and parsing.

8 Related work

We proposed a set of self-management ontologies which are not existing in the related pervasive computing ontologies, such as SOUPA and Amigo [12]. These self-management ontologies are the key to enable various self-management tasks. At the same time, we can model complex contexts using SWRL with the Hydra ontologies [19]. Work in [15] applied SWRL-based context modeling, and illustrated three cases of applying SWRL to manipulate context. We go beyond it by the dynamic-state based monitoring and diagnosis using the context ontologies.

In this paper, we largely followed a Layered architecture proposed by Kramer and Magee [14] for self-managed systems, which is composed of component control, change management and goal management, but mainly focus on the Component Control and Change Management. Our implementation adopts a mix of Blackboard architecture and Layered architecture to improve performance and extensibility.

Self-healing is one of the main challenges to autonomic pervasive computing. The basic process of state based diagnosis is using of states for detecting source of failure, and then notification of failure source and then resolve it. ETS [2] is following this idea, and so for our approach. Various failures in a pervasive system are classified in [5], and an architecture for fault tolerant pervasive computing is proposed. Our semantic web based approach provides a way of intelligent detection and resolution, which is not easily achievable by ETS and other approaches. We focus not only on device failure monitoring, but also on system level detection using the relationships of different state machine instances.

Work in [10] also use semantic web approach for achieving self-managing. Our approach is non-intrusive, SWRL rules are automatically executed using state machine instead of explicitly inserting sensor code to program, and is more suitable for the characteristics of pervasive devices.

There are many researches dealing with diagnosis using traditional artificial intelligence, e.g. [3]. These work is not utilizing the context ontologies that are already existing in pervasive systems. In our vision, the open world assumption in OWL/SWRL, and hence in our approach, is very well suited for the openness of the pervasive computing environment in a harmonious way, which automatically rejects the approaches using Prolog kind of rules that use close world assumption.

As surveyed in Ghosh's work [8], various strategies for self-healing are used in the literature. We are using probing and monitoring (as for FlamencoProbe and StateMachine) to detect something (component or service) amiss, and for the detection of other malfunction situations. Recovery planing will be based on ontology reasoning from Malfunction ontology, QoS ontology and Service ontology. Our semantic web based self-management approach are taking into the characteristics of the pervasive service environment.

9 Conclusions and future work

Self-management capabilities are important to achieve dependability in pervasive systems, and is a challenge for pervasive computing. In this paper, we propose a semantic web based self-management approach for pervasive service environments. A set of self-management ontologies are presented within a hybrid context management framework. To enable the self-management, we are adopting both application level reporting and network level reporting for the notification of runtime status.

The proposed semantic web based self-management is suitable for the openness nature of pervasive computing. As semantic web based context modeling is extensively used in pervasive computing, it is beneficial to uniformly make use of this for self-management purposes. The evaluations of the Hydra Diagnosis Manager in terms of extensibility, scalability, and performance, shows that the proposed self-management is effective for the Hydra purposes. It is interesting to note that rule grouping can greatly improve the performance, and we will make use of this feature more extensively in later iterations.

We will continue the implementation of the Goal Management layer of the three layered architecture, where a larger scale of application domains are going to be implemented in the coming Hydra integration work, includ-

ing health care domain, building automation domain. And clearly we need to add probability capabilities into the rules and models as the diagnosis needs to be more up to reality. At the same time we are working on QoS ontology rules and QoS-awareness service matching and service selection based on the SWRL rules. Further work on the full scope of self-management, such as self-adaptation, self-configuration based on OWL/SWRL ontologies are also under way.

Acknowledgments

The research reported in this paper has been supported by the Hydra EU project (IST-2005-034891).

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoes. Performance debugging for distributed systems of black boxes. In *Proceeding of SOSP'03*, pages 74–89. ACM, 2003.

[2] S. Ahmed, M. Sharmin, and S. Ahamed. ETS (Efficient, Transparent, and Secured) Self-healing Service for Pervasive Computing Applications. *International Journal of Network Security*, 4(3):271–281, 2007.

[3] R. Barco, L. Díez, V. Wille, and P. Lázaro. Automatic diagnosis of mobile communication networks under imprecise parameters. *Expert Systems With Applications*, 2007.

[4] L. Baresi, E. Di Nitto, and C. Ghezzi. Toward Open-World Software: Issues and Challenges. *COMPUTER*, pages 36–43, 2006.

[5] S. Chetan, A. Ranganathan, and R. Campbell. Towards fault tolerant pervasive computing. *Technology and Society Magazine, IEEE*, 24(1):38–44, 2005.

[6] A. Dey. Understanding and Using Context. *Personal and Ubiquitous Computing*, 5(1):4–7, 2001.

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

[8] D. Ghosh, R. Sharman, H. Raghav Rao, and S. Upadhyaya. Self-healing systemsŮsurvey and synthesis. *Decision Support Systems*, 42(4):2164–2185, 2007.

[9] K. M. Hansen, W. Zhang, and G. Soares. Ontology-enabled generation of embeddedweb services. In *The 20th International Conference on Software Engineering and Knowledge Engineering*, pages 345–350, Redwood City, San Francisco Bay, USA, Jul. 2008.

[10] A. R. Haydarlou, M. A. Oey, B. J. Overeinder, and F. M. T. Brazier. Use-case driven approach to self-monitoring in autonomic systems. *The Third International Conference on Autonomic and Autonomous Systems*, 2007.

[11] U. Hustadt. Do we need the Closed World Assumption in Knowledge Representation. *Proc. of the 1st Workshop KRDB*, 94.

[12] IST Amigo Project. Amigo middleware core: Prototype implementation and documentation, deliverable 3.2. Technical report, IST-2004-004182, 2006.

[13] J. Kephart and D. Chess. The Vision of Autonomic Computing. 2003.

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

[15] D.-J. Plas, M. Verheijen, H. Zwaal, and M. Hutschemaekers. Manipulating context information with swrl. *I/RS/2005/117, Freeband/A-MUSE/D3.12*, 2006.

[16] M. Satyanarayanan. Pervasive computing: vision and challenges. *Personal Communications, IEEE [see also IEEE Wireless Communications]*, 8(4):10–17, 2001.

[17] T. Strang and C. Linnhoff-Popien. A Context Modeling Survey. *Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp*, pages 34–41, 2004.

[18] W. Zhang and K. M. Hansen. 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, Jul. 2008.

[19] W. Zhang and K. M. Hansen. Towards self-managed pervasive middleware using owl/swrl ontologies. In *Fifth International Workshop on Modeling and Reasoning in Context (MRC 2008)*, pages 1–12, Delft, The Netherlands, Jun. 2008.

[20] H. Zimmermann. OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.

A.4 Paper 4: Towards Self-managed Pervasive Middleware using OWL/SWRL ontologies

Fifth International Workshop Modeling and Reasoning in Context. Held at HCP 08, Delft, The Netherlands, 8-12 June 2008

Towards Self-managed Pervasive Middleware using OWL/SWRL ontologies

Weishan Zhang and Klaus Marius Hansen

Department of Computer Science, University of Aarhus
Aabogade 34, 8200 Århus N, Denmark
{zhangws, klaus.m.hansen}@daimi.au.dk

Abstract. Self-management for pervasive middleware is important to realize the Ambient Intelligence vision. In this paper, we present an OWL/SWRL context ontologies based self-management approach for pervasive middleware where OWL ontology is used as means for context modeling. The context ontologies are incorporating the dynamic context information, including device and service run time information, which can then be used for running status checking and diagnosis, QoS monitoring, and further to achieve other self-management features, such as the self-configuration and self-adaptation. We demonstrate the OWL/SWRL context ontologies based self-management approach with the self-diagnosis in Hydra middleware, using device state machine and other dynamic context information, for example web service calls. The evaluations in terms of extensibility, performance and scalability show that this approach is effective in pervasive service environment.

1 Introduction and Motivation

Context awareness[1] is one of the key features for pervasive middleware. It can provide the potential to improve the flexibility and personality during service provision, and alleviate the human attention and interaction bottlenecks by providing self-management features using contexts, including self-configuration, self-adaptation, self-optimization, self-protection and self-healing (through self-diagnosis). This is vital to achieve the vision of Ambient Intelligence (AmI) that should come with the pervasive middleware like Hydra (IST-2005-034891).

To facilitate achieving context-awareness, we agree that OWL[1] ontology is the best way for context modeling [2], which can provide reasoning potentials for what contexts we are in, a capability not easily achievable by other context modeling approaches. The definition of *context* in [1] is general enough to cover the contexts in pervasive computing, but we want to point out that not only static knowledge, but also dynamic and runtime context should be considered in order to handle runtime-related requirements. For example, we can run a status check of a system at runtime, and monitor the dynamic contexts of the system and then make decisions on where the problem is, why the problem happens and how to tackle the problem.

[1] OWL homepage. http://www.w3.org/2004/OWL/.

Take a concrete agriculture scenario in the Hydra project:
Bjarne is an agricultural worker at a large pig farm in Denmark. As he checks whether the pigs are provided with the correct amount of food, he is interrupted by a sound from his PDA. Apparently, the ventilation system in the pig stable has malfunctioned. After acknowledging the alarm, the system begins to diagnosis and soon it decides that the cause of the problem is 'power supply off because of fuse blown'. Then he can prepare a fuse and repair the ventilator. After repairing it, he signs off the alarm, and chooses one of the predefined log messages, describing what he has done.

As can be seen from the above scenario, it is very important that the Hydra middleware can provide diagnosis functionality to the end user, or better to achieve self-healing when there is malfunction. To this end, the run time information, for example the device states should be monitored in order to make decisions on diagnosis. Other self-managing work for the Hydra middleware among others includes self-protection according to security rules, searching service and negotiating QoS (Quality of Service) parameters with other services.

In this paper, we will show the Hydra context ontologies that considering run time contexts, and present an OWL ontology and SWRL (Semantic Web Rule Language)[2] based self-management approach for Hydra, in particular the self-diagnosis, which take into the characteristics of pervasive computing. We demonstrate our approach with the Diagnosis Manager for Hydra middleware. The evaluations in terms of extensibility, performance, and scalability shows that the proposed Hydra OWL/SWRL context ontologies and self-management approach based on OWL/SWRL context ontologies are effective to achieve the self-diagnosis goals, and lay a solid foundation for other self-management work.

The rest of the paper is structured as follows: in Section 2 we will briefly introduce the architecture of self-management based on OWL/SWRL ontologies of the Hydra middleware; We then show the Hydra ontologies that facilitate self-management, followed in Section 4 we present the complex context specification with SWRL; In Section 5, we demonstrate the proposed approach with the Diagnosis Manager together with some evaluations. We compare our work with the related work in Section 6. Conclusions and future work end the paper.

2 Architecture of self-management in Hydra

The Hydra project is developing self-managed middleware for pervasive embedded and network systems based on service-oriented architecture. Several components are involved in achieving the self-management features, based on context ontologies where dynamic contexts are encoded. These components include a Diagnosis Manager, which is used to monitor the system conditions and states in order to fulfill error detection, diagnosis, and provide recovery solutions; and a QoS Manager negotiating QoS parameters with other services and manages resources accordingly. Further, the QoS Manager provides device specific infor-

[2] SWRL specification homepage. http://www.w3.org/Submission/SWRL/

mation to the Diagnosis Manager, and coordinates with Service Manager, Ontology Manager and Orchistration Manager. Context events are managed using an Event Manager where publish/subscribe functionalities are provided.

To build necessary intelligence into the Hydra middleware in order to support the self-management, the related monitoring and dianosis rules, QoS rules and service selection rules built on top of the ontologies, play a vital role. We chose SWRL to develop these rules. SWRL is a W3C recommendation for the rule language of the Semantic Web, which can be used to write rules to reason about OWL individuals and to infer new knowledge about those individuals. SWRL provides builtins such as math, string and comparisons that can be used to specify extra contexts, which are not possible or very hard to achieve by OWL itself. Therefore, SWRL is naturally chosen as the rule language in Hydra for the implementation of self-management rules.

The architecture for all the self-management components is the same for Hydra. We are following the three Layered architecture proposed by Kramer and Magee [3]. Based on the current status of OWL/SWRL, we came up with the following architecture as shown in Figure 1, in which the *Goal Management*, *Component Control* and *Change Management* are enclosed with dashed line. The bottom of the architecture is the ontologies/rules, in which knowledge of devices, rule based QoS, and state based diagnosis are encoded. For the Component Control layer, it is mainly used for state reporting and run time information updating, for example battery level and QoS measurement. For the Change Management layer, it is used to response to the state reported from the Component Control layer, and then execute rules developed based on these state and other run time information. For the Goal Management layer, it is used to resolve the rule conflicts based on QoS regulations or user preference etc.

3 Context ontologies in Hydra

Context-awareness, especially the awareness of those dynamic context information, is the most important factor to fulfill the goal of various self-management processes. In Hydra, the context awareness has the following awareness features:

- *Resource awareness* This includes hardware, for example CPU, and software, for example operating system.
- *Power awareness* Different network carriers use different amount of energy during transmission. This will be considered during service provision. Battery information for device is also need to be known.
- *QoS awareness* As one of the important criteria for the selection of service, QoS is another context that shows both static and dynamic affects to the middleware, for example latency.
- *Security awareness* The right information should be transferred to the right user at the right time in the right place using the agreed service level agreement, and in the appropriate format.

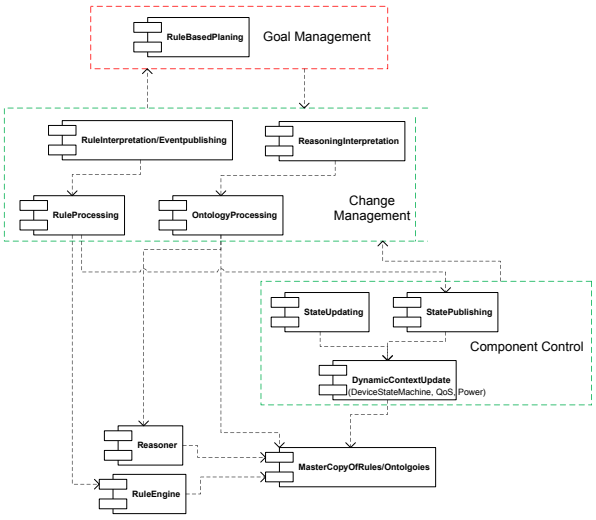

Fig. 1. Architecture of the self-management in Hydra

3.1 Structure and design of the Hydra ontologies

Although there are some pervasive computing ontologies, e.g. SOUPA[3] ontologies, they are not enough for achieving the needed intelligence and handling of dynamic context in order to achieve the above mentioned various awareness. The openness and dynamism of pervasive computing, and the nature for pervasive and embedded devices running as state machines, motivate the development of Hydra context ontologies, whose high level structure is shown in Figure 2.

The Device ontology itself is used to define some basic information of a Hydra device, for example device type classification(e.g. mobile phone, PDA, sensor), device model and manufacturer, and so on. The device type classification in the Device ontology is based mainly on AMIGO project ontologies [4]. To facilitate diagnosis, there is a concept called *HydraSystem* to model a system composed of devices to provide services. And there is a corresponding object property *hasDevice* which has the domain of *HydraSystem* and range as *HydraDevice*. There are also concepts used for the monitoring of web service calls, including SocketProcess, SocketMessage and IPAddress. The *HydraDevice* concept has a data-type property *currentMalfunction* which is used to store the inferred device malfunction diagnosis information at run time and will be exemplified later.

The device Malfunction ontology is used to model knowledge of malfunction and recovery resolutions. We separate the malfunctions into two categories: *Error* (including device totally down) and *Warning* (including function scale-down, and plain warning). There are also two other concepts, *Cause* and *Remedy*, which are used to describe the origin of a malfunction and its resolution.

[3] Semantic Web in Ubicomp. http://pervasive.semanticweb.org/.

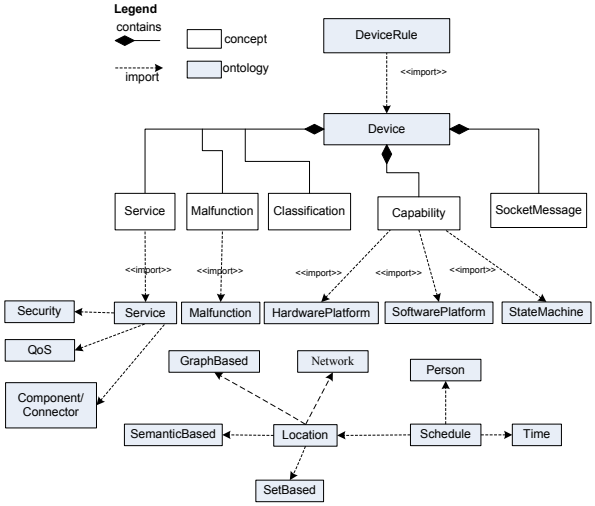

Fig. 2. Structure of the Hydra context ontologies

3.2 Dynamic context

The dynamic context information will reflect the running status of the underlying system, therefore it is the key enabler for the functioning of self-management. For example, the device status is important to check the device working condition and is used to develop diagnosis and monitoring rules; the monitoring of QoS is important to test whether service level agreement is meet due to service mobility.

A common sense of mobile and embedded devices used in pervasive environments is that they are usually designed and operated as state machines. In line with this, a state machine ontology is developed based on [5] with many improvements:

- A data-type property *isCurrent* is used in order to indicate whether a state is current or not.
- A *doActivity* object property is added to the *State* in order to specify the corresponding activity in a state and this makes the state machine complete.
- A data-type property *hasResult* is added to the *Action* (including activity) concept in order to check the execution result at runtime.
- Three data-type properties are added to model historical action results.

The dynamic context is modeled with runtime concepts and properties in the related ontologies, mainly the StateMachine ontology, the Malfunction ontology, QoS ontology, and other concepts and properties in the Device ontology, such as *currentMalfunction* and *HydraSystem*. The *currentMalfunction* will be used to store the current diagnosis information for the malfunction case, *HydraSystem* is used to dynamically model the device joining and leaving and reflect the composition of a system.

Because of the space limit, in this paper we only present the self-diagnosis to demonstrate the effectiveness of the self-management using these ontologies. Figure 3 shows a more detailed but simplified view of the ontologies facilitating diagnosis.

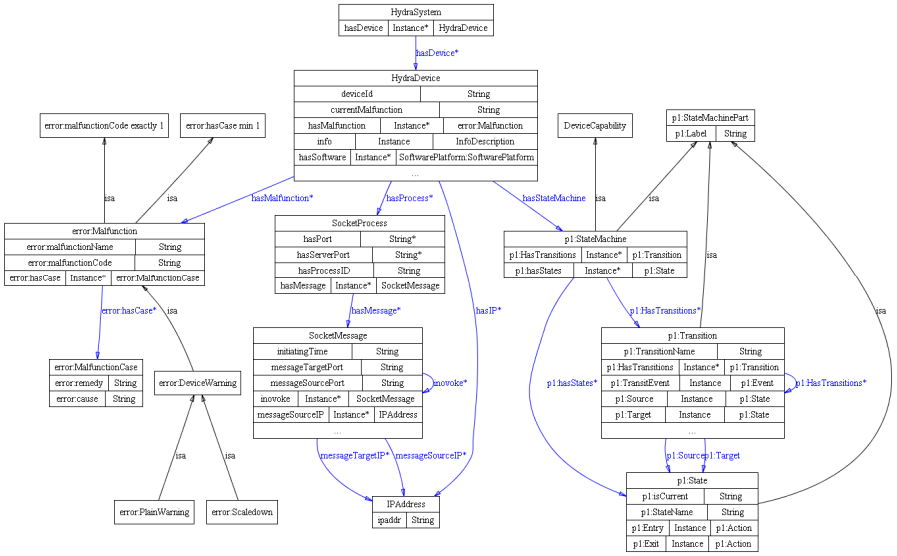

Fig. 3. Partial details of the Diagnosis Manager used ontologies

4 Extending OWL ontologies with SWRL rules

A SWRL rule is composed of an antecedent part (body), and a consequent part (head). Both the body and head consist of positive conjunctions of atoms. A SWRL rule means that if all the atoms in the antecedent (body) are true, then the consequent (head) must also be true. SWRL is built on OWL DL and shares its formal semantics. In our practice, all variables in SWRL rules bind only to known individuals in an ontology in order to develop DL-Safe rules to make them decidable. In our example SWRL rules, the symbol $\wedge$ means conjunction, and $?x$ stands for a variable, $\rightarrow$ means implication, and if there is no ? in the variable, then it is an instance.

4.1 Complex context specification with SWRL rules

SWRL has more expressive power than OWL using various builtins, and can be used to specify complext contexts. As an example, we can specify a GPS distance

calculation with SWRL in order to define a *farAwayFromHome* context (e.g. 5 miles away from home using the GPS distance calculation formula[4]). Then this new context can be used to take actions, for example, surveillance system is switched automatically to the highest security level with all cameras turned on.

$person : hasHome(?person, ?home) \wedge$
$person : inLocation(?person, ?coord1) \wedge$
$loc : hasCoordinates(?home, ?coord2) \wedge$
$coord : latitude(?coord1, ?lan1) \wedge$
$coord : latitude(?coord2, ?lan2) \wedge$
$swrlb : subtract(?sub1, ?lan1, ?lan2) \wedge$
$swrlb : multiply(?squaresublan, ?sub1, ?sub1) \wedge$
$swrlb : multiply(?par1, ?squaresublan, 4774.81) \wedge$
$coord : longitude(?coord1, ?long1) \wedge$
$coord : longitude(?coord2, ?long2) \wedge$
$swrlb : subtract(?sub2, ?long1, ?long2) \wedge$
$swrlb : multiply(?squaresublong, ?sub2, ?sub2) \wedge$
$swrlb : multiply(?par2, ?squaresublong, 2809) \wedge$
$swrlb : add(?parameter, ?par1, ?par2) \wedge$
$swrlm : sqrt(?distance, ?parameter) \wedge$
$swrlb : greaterThan(?distance, 5) \wedge$
$swrlb : StringConcat(?str, '' true'')$
$\rightarrow sqwrl : select(?person, ?home, ?distance) \wedge farAwayFromHome(?person, ?str)$

Similarly, we can define QoS metrics and other QoS regulation with SWRL rules, and we are investigating the specifying of service selection using SWRL rules based on the Security ontology, Service ontology and QoS ontology. Here is a simple example of querying the availability dynamically.

$swrlb : add(?total, p1 : downtime, p1 : uptime) \wedge$
$swrlb : divide(?availability, p1 : uptime, ?total)$
$\rightarrow sqwrl : select(?availability)$

4.2 Complex dynamic context

To achieve self-management, for example self-diagnosis, the OWL context ontologies themselves are really weak to specify rules that are important to define policies for security, QoS metric calculation and diagnosis rules. In these cases, we are applying SWRL to specify these dynamic contexts.

For this paper, we will elaborate on the self-diagnosis contexts which rely on the device state machine and other related concepts as mentioned in the former section. In a similar way, the QoS monitoring rules could be developed based on the dynamic information monitored.

Monitoring and diagnosis rules are the basis for the diagnosis service. We have two level diagnosis rules, namely device level rules and system level rules. Device level rules are used for a certain type of devices which are supposed to be

[4] How to calculate the distance between two points on the Earth. http://www.meridianworlddata.com/Distance-Calculation.asp

generic for that type of devices. The following example rule specifies the mobile phone battery level monitoring. If the battery level is less than 10%, then a warning will be published.

$device : MobilePhone(?device) \wedge$
$device : hasHardware(?device, ?hardware) \wedge$
$Hardware : primaryBattery(?hardware, ?battery) \wedge$
$Hardware : batteryLevel(?battery, ?level) \wedge$
$swrlb : lessThanOrEqual(?level, 0.1)$
$\rightarrow VeryLowBattery(?device)$

System level rules are used to specify rules that span multiple devices in a system. For example, if the detected flow for feeding the pig is more than 6 gallon per minute, then we can infer that the pipe is broken.

$device : FlowMeter(?device) \wedge$
$device : hasStateMachine(?device, ?statemachine) \wedge$
$statemachine : hasStates(?statemachine, ?state) \wedge$
$statemachine : doActivity(?state, ?action) \wedge$
$statemachine : actionResult(?action, ?result) \wedge$
$abox : isNumeric(?result) \wedge$
$swrlb : greaterThan(?result, 6.0) \rightarrow device : currentMalfunction(device : Flowmeter, "PipeBroken")$

A more complex rule is the example of pig farm ventilating and monitoring system, thermometers are used to measure both indoor and outdoor temperature. In the summer time, the indoor temperate should follow the same trend as the outdoor temperature. A rule is developed to first obtain the trends by the difference of continuous temperature measurements of both the indoor and outdoor temperatures. If the trend is not the same, we infer that the ventilator is down.

From our experiences, the loading of the OWL/SWRL ontologies is the main performance bottleneck, therefore all the current rule sets are stored in one separate ontology called *DeviceRule* as can be seen from Figure 2, and we load it when the system initializes.

5 Evaluating OWL/SWRL context ontologies based self-management with Diagnosis Manager

To evaluate the OWL/SWRL context ontologies based self-management approach, and the proposed Hydra ontologies, we will use the Diagnosis Manager as an example to show the effectiveness and usability, in terms of extensability, performance (as the main concern), and scalability.

5.1 Design and implementation of the Diagnosis Manager

Hydra implements a service-oriented architecture based on web service interaction among devices. Initially we focus on device status reporting using state

changes as events through the Hydra Event Manager, and Web service request/reply reporting using IPSniffer tool. When there are state change events, the device state machine instance in the state machine ontology need to be updated, and also these state changes will be published with state machine state changes as event topic. The Diagnosis Manager is an event subscriber to the state machine state change events, it will then update the corresponding state instances in the ontology. At the same time, this will trigger the diagnosis of the device status, executing the SWRL rules to monitor the health status of devices, and also trigger the reasoning of possible device errors and their resolutions. The Diagnosis Manager will publish the diagnosis results as an event publisher.

We adopted a mix of the Blackboard architecture style and the Layered architecture in the actual implementation due to the high overhead for loading ontologies, and use the observer pattern in both the updating of state machine ontology and parsing fo the inferred result from SWRL rules.

OWL ontology provides intelligence capabilities for diagnosis decisions. For example, Bjarne get a high priorit warning of "GrundfosPumpMQ345 failed to start". A diagnosis task is initiated to check what is wrong with the pump, but as a newly installed pump, there is still no error resolution to this model of pump in the Malfunction ontology. As a further step, the diagnosis system will conduct subsumption reasoning and search for the device *Type* in the Device ontology, which is found as *FluidPump*, and then its manufacturer is also queried. Now another query to the Device ontology will get a similar pump called *GrundfosPumpMQ335* as of the same type from the same manufacturer "Grundfos". And based on the name of the error and pump type, the solution from a query to Malfunction ontology is suggested "replace a capacitor", which is happily the solution to solve the problem.

5.2 Evaluation of the Diagnosis Manager

We evaluated the extensibility of the OWL/SWRL based diagnosis Manager in terms of scalability, performance, and extensibility. We started the development of Diagnosis Manager with the rule for temperature monitoring. After finishing the implementation and testing, we then try to handle the flowmeter diagnosis rules. We only need to add the flowmeter rules to the existing rules set. No single line of Diagnosis Manager code needs to be changed. In summary, the Diagnosis Manager has good extensibility.

For the measurement of performance, the following software platform is used: Protege 3.4 Build 125, JVM 1.6.02-b06, Heap memory is 266M, Windows Vista. The hardware platform is: Thinkpad T60 Core2Duo 2G CPU, 7200rpm hard disk, 2G DDR2 RAM. The time measurement is in millisecond. The size of DeviceRule ontology is 210,394 bytes, and contains 22 rules, which is fair for a small pervasive system in which monitoring and diagnosis functions are all included. The performance figures are shown in Table 1. An interesting thing is after some time of running, the Diagnosis Manager is running stably with the total time in 260-270 ms for processing an event, a bit faster than the one when

it starts. Here the parsing of the inferred result is running in a multi-threaded way in the Diagnosis Manager.

Update	InferringTime	AfterEventTillInferred
383	380	382
322	319	321
282	278	282
272	269	271
265	263	265
270	267	269
268	266	269

Table 1. Diagnosis Manager performance

For scalability, a number of events are published (almost in parallel) to measure how long it will be, starting from the publishing till the end of inferring and publish related inferring result. Time needed (y-axis) is shown in Figure 4 (x-axis shows the number of events) . We can see that the time taken is in linear with the events need to be processed.

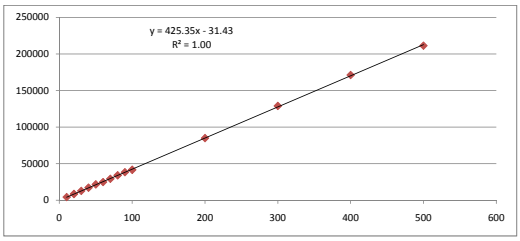

Fig. 4. Diagnosis Manager scalability

6 Related work

The work on *Context OWL*[6] considers that contexts are *local*. In the pervasive computing environment, this is not always true as we have a global *Time* manager which manages the consistent time for all users. Work in [7] also applied SWRL-based context modeling, and illustrated three cases of applying SWRL to manipulate context. We go beyond this work by the dynamic-state based monitoring and diagnosis using the context ontologies. When compared to the existing pervasive computing ontologies, such as SOUPA and Amigo[4], the Hydra context ontologies have some unique features. Firstly, dynamic contexts are incorporated which facilitate the achieving of self-management. Secondly, complex contexts, especially those self-management needed complex and dynamic

contexts, are defined by SWRL, which are not expressible by OWL ontology itself.

Kramer and Magee [3] recently proposed a reference model for self-managed systems, which is composed of component control, change management and goal management. In this paper, we largely followed this work for the Layered architecture, but mainly focus on the Component Control and Change Management. At the same time, a mix of Blackboard architecture and Layered architecture are applied to improve performance and extensibility.

Self-healing is one of the main challenges to autonomic pervasive computing. Generally speaking, our approach applied the same idea of ETS [8], in terms of the using of states for detecting source of failure, and then notification of failure source. And this process is actually universal for error detection. Our ontology and SWRL rule based approach provides a way of intelligent detection and resolution, which is not easily achievable by ETS.

Work in [9] also use semantic web approach for achieving self-managing. Our approach is non-intrusive, SWRL rules are automatically executed using state machine instead of explicitly inserting sensor code to program, and is more suitable for the characteristics of pervasive devices.

Various failures in a pervasive system are classified in [10], and an architecture for fault tolerant pervasive computing is proposed. We focus not only on device failure monitoring, but also on system level detection using the relationships of different state machine instances. In addition, our approach can be more intelligent in terms that ontology reasoning can help the diagnosis.

There are many researches dealing with the diagnosis in various field, e.g. [11] from traditional artificial intelligence point of view. These traditional approaches are not utilizing the context ontologies that are already there in pervasive systems and are used for context-awareness and other purposes. In our vision, the open world assumption in OWL/SWRL, and hence in our approach, is very well suited for the openness of the pervasive computing environment, which automatically rejects the approaches using Prolog kind of rules that use close world assumption.

7 Conclusions and future work

Self-management capabilities are important to achieve necessary dependability in pervasive system, and is a challenge for pervasive computing. In Hydra, we make use of the OWL/SWRL ontologies as the basis for the implementation of self-management features, which is very suitable for the openness nature of pervasive computing. These context ontologies are incorporating the dynamic context information, including device and service run time information, which can then be used for running status checking and diagnosis, QoS monitoring, and further to achieve other self-management features, such as the self-configuration and self-adaptation.

We illustrate the OWL/SWRL based self-management with the experiment of the Diagnosis Manager, mainly using state machine ontology and SWRL rules

built based on it. The malfunction information and its resolution are encoded in an OWL ontology, and can be used at run time to infer the solution to the malfunction, and further to fulfill self-healing activities. SWRL is used to develop monitoring and diagnosis rules, which can help make intelligent decisions when there is malfunction occurs. The evaluations of the Diagnosis Manager in terms of extensibility, scalability, and performance, relieved us for the worrying of performance of the OWL/SWRL based self-management.

In the future, we will continue the implementation of the Goal and planning layer of the three layered architecture. At the same time we are working on QoS ontology rules and QoS-awareness service matching and service selection based on the SWRL rules. Further work on full scope of self-management, such as self-adaptation, self-configuration based on OWL/SWRL ontologies are under way, which will be reported in the coming papers.

Acknowledgements

The research reported in this paper has been supported by the Hydra EU project (IST-2005-034891).

References

1. Dey, A.: Understanding and Using Context. Personal and Ubiquitous Computing **5**(1) (2001) 4–7
2. Strang, T., Linnhoff-Popien, C.: A Context Modeling Survey. Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp (2004) 34–41
3. Kramer, J., Magee, J.: Self-Managed Systems: an Architectural Challenge. International Conference on Software Engineering (2007) 259–268
4. IST Amigo Project: Amigo middleware core: Prototype implementation and documentation, deliverable 3.2. Technical report, IST-2004-004182 (2006)
5. Dolog, P.: Model-driven navigation design for semantic web applications with the uml-guide. Engineering Advanced Web Applications, In Maristella Matera and Sara Comai (eds.) (Dec. 2004)
6. Bouquet, P., Giunchiglia, F., van Harmelen, F.e.a.: C-OWL: Contextualizing ontologies. Second International Semantic Web Conference (2003) 164–179
7. Plas, D.J., Verheijen, M., Zwaal, H., Hutschemaekers, M.: Manipulating context information with swrl. I/RS/2005/117, Freeband/A-MUSE/D3.12 (2006)
8. Ahmed, S., Sharmin, M., Ahamed, S.: ETS (Efficient, Transparent, and Secured) Self-healing Service for Pervasive Computing Applications. International Journal of Network Security **4**(3) (2007) 271–281
9. Haydarlou, A.R., Oey, M.A., Overeinder, B.J., Brazier, F.M.T.: Use-case driven approach to self-monitoring in autonomic systems. The Third International Conference on Autonomic and Autonomous Systems (2007)
10. Chetan, S., Ranganathan, A., Campbell, R.: Towards fault tolerant pervasive computing. Technology and Society Magazine, IEEE **24**(1) (2005) 38–44
11. Barco, R., Díez, L., Wille, V., Lázaro, P.: Automatic diagnosis of mobile communication networks under imprecise parameters. Expert Systems With Applications (2007)

A.5 Paper 5: An OWL/SWRL based Diagnosis Approach in a Pervasive Middleware

Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE'2008), San Francisco, CA, USA, July 1-3, 2008

An OWL/SWRL based Diagnosis Approach in a Pervasive Middleware

Weishan Zhang and Klaus Marius Hansen
Department of Computer Science, University of Aarhus
Aabogade 34, 8200 Århus N, Denmark
{zhangws,klaus.m.hansen}@daimi.au.dk

Abstract

Diagnosis is the most important step for achieving self-healing of systems, which is a challenge in pervasive computing. In this paper, we present a semantic, state machine-based diagnosis approach for a web-service based middleware. We use OWL ontologies and SWRL to develop both diagnosis and monitoring rules, based on state changes and also invocation relationships. Malfunction information and its resolution are encoded in an OWL ontology as a part of a Device ontology, and can be used at run time to check how to resolve malfunction, and further to fulfill self-healing activities. SWRL rules at both device level and system level are designed and will be executed as needed. The evaluations in terms of extensibility, performance and scalability show that this approach is effective in pervasive service environment.

1 Introduction and Motivation

Web services are increasingly needed to be adopted as service provision mechanisms in pervasive computing environment. This trend is exemplified during the inauguration phase of the *Hydra* project(IST-2005-034891), by some companies that donate us Zigbee devices and other embedded devices that enabling pervasive computing, and express their wishes for web service enabled devices.

A concrete agriculture scenario that we are considering in the Hydra project is as followed:
Bjarne is an agricultural worker at a large pig farm in Denmark. As he walks through the pens to check whether the pigs are provided with correct amount of food, his work is interrupted by a sound from his PDA, indicating that a high priority alarm has arrived. Apparently, the ventilation system in the pig stable has malfunctioned. After acknowledging the alarm and the system begins to diagnosis and soon it decides that the cause of the problem is 'power supply off because of fuse blown'. Then he can prepare a fuse and repair the ventilator. After repairing it, he signs off the alarm, and writes a log on what he has done.

As can be seen from the above scenario, it is very important that the Hydra middleware can provide diagnosis functionality to the end user, or better to achieve self-healing when there is malfunction. Such kind of self-healing can not be always finished automatically, for example device down because of fuse broken. But providing diagnosis and then resolution suggestions would be the most important step towards malfunction recovery.

In this paper, we present an OWL ontology (the Web Ontology language)[1] and SWRL (Semantic Web Rule Language)[2] based diagnosis using state machine and sniffering of process invocation in the context of the Hydra middleware. The malfunction information and its resolution are encoded in an OWL ontology as part of a Device ontology, and can be used at run time to check appropriate resolution to the malfunction, and further to fulfill self-healing activities. We use SWRL to develop monitoring and diagnosis rules, and these rules, together with OWL ontologies, can help make intelligent decisions on where malfunction occurs and its resolution.

The rest of the paper is structured as follows: Section 2 presents an overview of the Hydra middleware; We then show the diagnosis ontologies used in Hydra; In section 4, design of both rules and the Diagnosis Manager are presented. Section 5 evaluates our work with the extensibility, performance and scalability. We compare our work with the related work in section 6. Conclusions and future work end the paper.

2 Web service based middleware-Hydra

The Hydra project is developing a service-oriented and self-managed middleware for pervasive embedded and network systems based on web service. According to the available resources, the function structure of the Hydra middleware is divided into two parts, namely Application El-

[1] OWL Web Ontology Language Guide. http://www.w3.org/TR/owl-guide/

[2] SWRL specification homepage. http://www.w3.org/Submission/SWRL/

ements(AEs) and Device Elements(DEs). AEs are meant to be running on powerful machines, DEs describe components that are usually deployed inside Hydra-enabled devices where small devices maybe involved. The Layered architecture of the Hydra middleware is shown in Figure 1.

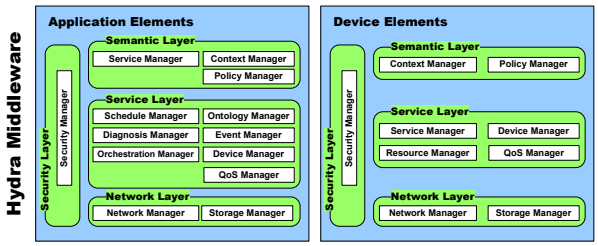

Figure 1. Hydra middleware Layered architecture

Diagnosis Manager is used to monitor the system conditions and states in order to fulfill error detection and logging device events. Its functions include system diagnosis and device diagnosis.

The Event Manager is used to provide publish/subscribe functionality to the HYDRA middleware. In general, publish/subscribe communication as provided by the Event Manager provides an application-level, selective multicast that decouples senders and receivers in time, space and data.

3 Ontologies used in the Diagnosis Manager

There are several ontologies involved in the diagnosis process, namely Device ontology, Malfunction ontology, and StateMachine ontology. The DeviceRule ontology is used for holding all diagnosis rules as introduced in Section 4.1. The high level structure of the diagnosis ontologies is shown in Figure 2.

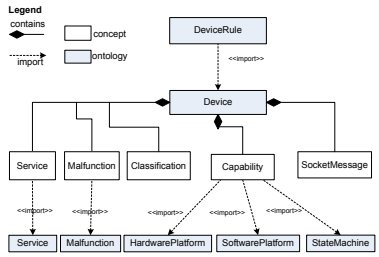

Figure 2. Diagnosis ontologies structure

The Device ontology is used to define some basic information of a Hydra device, for example device type classification(e.g. mobile phone, sensor), device model and manufacturer, and so on. The device type classification is based mainly on AMIGO project ontologies [7]. To facilitate diagnosis, there is a concept called *HydraSystem* to model a system composed of devices to provide services. And there is a corresponding object property *hasDevice* which has the domain of *HydraSystem* and range as *HydraDevice*. There are also concepts used for the monitoring of web service calls, including *SocketProcess*, *SocketMessage* and *IPAddress*. The *HydraDevice* concept has a data type property *currentMalfunction* which is used to store the inferred device malfunction diagnosis information at run time and will be exemplified later.

To enable state based diagnosis, a state machine ontology is developed based on [5] with many improvements: firstly, we add a datatype property *isCurrent* in order to indicate whether a state is current or not; secondly, we add a *doActivity* object property to the *State* in order to specify the corresponding activity in a state and this makes the state machine complete; thirdly, we add a datatype property *hasResult* to the *Action* (including activity) concept in order to check the execution result at run time. Three other datatype properties are also added to model historial action results. This facilitates the specification of diagnosis rule based on state and activity result and its history.

The device Malfunction ontology is used to model malfunction and recovery resolutions. We separate the malfunctions into two categories: *Error* (including device totally down) and *Warning* (including function scale-down, and plain warning). There are also two other concepts, *Cause* and *Remedy*, which are used to describe the origin of malfunction and its resolution.

A more detailed but simplified view of the ontologies used in the diagnosis is depicted in Figure 3.

4 Design of the Diagnosis Manager

Hydra implements a service-oriented architecture based on web service interaction among devices. Thus a reasonable granularity to build a self-management system on is the level of web service requests and responses. Furthermore, we are interested in the states of devices per se, i.e., is the device operational, stopped, not working and if it is operational what is the value of its sensor readings (if any) or its actuator state (if any). This leads us initially to focus on status reporting of the following two forms:

- *State change reporting.* State machines are used to report their state changes as events through the Hydra Event Manager.
- *Web service request/reply reporting.* The requests and replies (and their associated data) can be used to analyse the runtime structure of the Hydra systems. Here a tool called IPSniffer is used to report invocations.

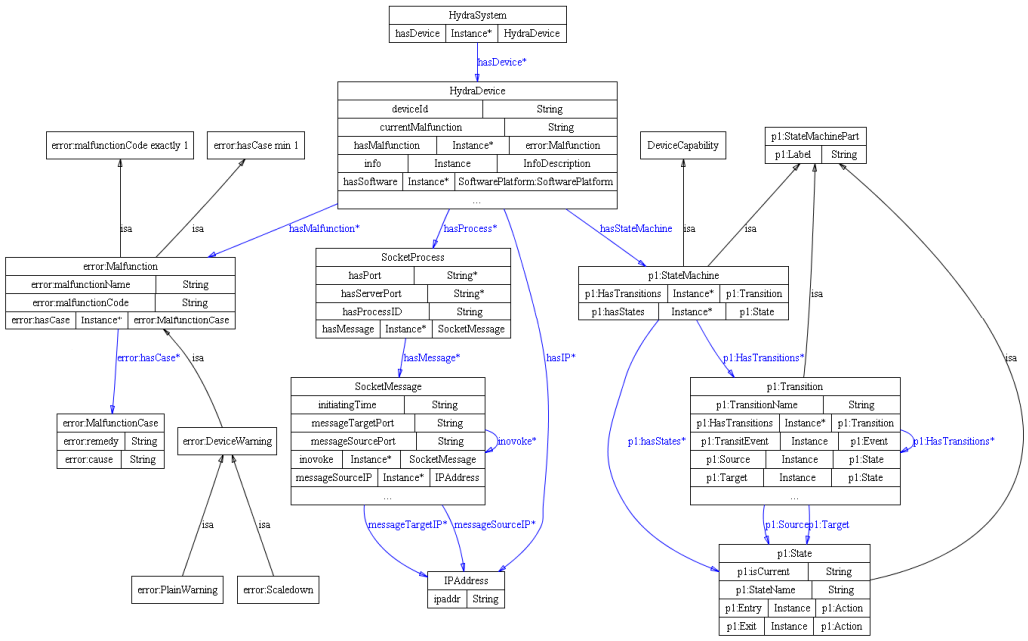

Figure 3. Partial details of the Diagnosis Manager used ontologies

4.1 Design of SWRL rules

Diagnosis is a complex task which need intelligence to infer what is the reason for error and its consequence. The OWL-DL ontologies themselves are hardly expressive enough to specify diagnosis rules. As an alliance to OWL, SWRL can be used to write rules to reason about OWL individuals and to infer new knowledge about those individuals. A SWRL rule means that if all the atoms in the antecedent (body) are true, then the consequent (head) must also be true. In the SWRL rules, the symbol $\wedge$ means conjunction, $?x$ stands for a variable, $\rightarrow$ means implication; and if there is no ? in the variable, then it is an instance.

Device level rules

Device level rules are used for a certain type of devices which are supposed to be generic for that type of devices. The followed is an example of mobile phone battery monitoring, if battery level is less than 10%, a warning will be published.

$device : MobilePhone(?device) \wedge$
$device : hasHardware(?device, ?hardware) \wedge$
$Hardware : primaryBattery(?hardware, ?battery) \wedge$
$Hardware : batteryLevel(?battery, ?level) \wedge$
$swrlb : lessThanOrEqual(?level, 0.1)$
$\rightarrow VeryLowBattery(?device)$

Another monitoring rule is if the flow measured from the flowmeter is more than 16 (gallon/minute), then it is too high and should be repaired as soon as possible:

$device : FlowMeter(?device) \wedge$
$device : hasStateMachine(?device, ?sm) \wedge$
$statemachine : hasStates(?sm, ?state) \wedge$
$statemachine : doActivity(?state, ?action) \wedge$
$statemachine : actionResult(?action, ?r) \wedge$
$abox : isNumeric(?r) \wedge swrlb : greaterThan(?r, 16.0) \rightarrow$
$device : currentMalfunction(device : Flowmeter, "FlowHigh")$

The rule for IPSniffer is used for both checking process id, ip address, port etc. and inferring invoking relationships.

$device : messageSourceIP(?message1, ?ip1) \wedge$
$device : ipaddr(?ip1, ?ipa1) \wedge$
$device : messageSourcePort(?message1, ?port1) \wedge$
$device : hasMessage(?process1, ?message1) \wedge$
$device : hasProcessID(?process1, ?pid1) \wedge$
$device : messageTargetIP(?message2, ?ip2) \wedge$
$device : messageSourceIP(?message2, ?ip3) \wedge$
$device : ipaddr(?ip3, ?ipa3) \wedge$
$device : messageTargetPort(?message2, ?port2) \wedge$
$device : hasMessage(?process2, ?message2) \wedge$
$device : hasProcessID(?process2, ?pid2) \wedge$
$swrlb : equal(?port1, ?port2) \wedge$
$device : initiatingTime(?message1, ?time1) \wedge$
$device : initiatingTime(?message2, ?t2) \wedge$
$temporal : duration(?d, ?time1, ?t2, temporal : Milliseconds)$
$\wedge swrlb : lessThanOrEqual(?d, 60000)$
$\rightarrow device : inovoke(?message1, ?message2) \wedge$

$sqwrl : select(?ipa1, ?port1, ?pid1, ?ipa3, ?port2, ?pid2, ?time1)$

System level rules

System level rules are used to specify rules span multiple devices in a system. In the introduced agriculture scenario, thermometers are used to measure both indoor and outdoor temperature, which are named PicoTh03_Outdoor and PicoTh03_Indoor respectively. In the summer time, when outdoor temperature is between 12 and 33 degree, the indoor should follow the same trend as the outdoor temperature. Or else, we can infer that the ventilator is down.

$device : hasStateMachine(device : PicoTh03_{O}utdoor, ?sm)$
$\wedge\ statemachine : hasStates(?sm, ?state) \wedge$
$statemachine : doActivity(?state, ?action) \wedge$
$statemachine : actionResult(?action, ?r) \wedge$
$statemachine : historicalResult1(?action, ?r1) \wedge$
$statemachine : historicalResult2(?action, ?r2) \wedge$
$statemachine : historicalResult3(?action, ?r3) \wedge$
$swrlb : add(?tempaverage, ?r1, ?r2, ?r3) \wedge$
$swrlb : divide(?average, ?tempaverage, 3) \wedge$
$swrlb : subtract(?temp1, ?r, ?r1) \wedge$
$swrlb : subtract(?temp2, ?r1, ?r2) \wedge$
$swrlb : subtract(?temp3, ?r2, ?r3) \wedge$
$swrlb : add(?temp, ?temp1, ?temp2, ?temp3) \wedge$
$swrlb : greaterThan(?average, 12.0) \wedge$
$swrlb : lessThan(?average, 33.0) \wedge$
$swrlb : lessThan(?temp, 0) \wedge$
$device : hasStateMachine(device : PicoTh03_{I}ndoor, ?sm_b)$
$\wedge\ statemachine : hasStates(?sm_b, ?state_b) \wedge$
$statemachine : doActivity(?state_b, ?action_b) \wedge$
$statemachine : actionResult(?action_b, ?r_b) \wedge$
$statemachine : historicalResult1(?action_b, ?r1_b) \wedge$
$statemachine : historicalResult2(?action_b, ?r2_b) \wedge$
$statemachine : historicalResult3(?action_b, ?r3_b) \wedge$
$swrlb : subtract(?temp1_b, ?r_b, ?r1_b) \wedge$
$swrlb : subtract(?temp2_b, ?r1_b, ?r2_b) \wedge$
$swrlb : subtract(?temp3_b, ?r2_b, ?r3_b) \wedge$
$swrlb : add(?temp_b, ?temp1_b, ?temp2_b, ?temp3_b) \wedge$
$swrlb : greaterThan(?temp_b, 0) \rightarrow device : currentMalfunction(device : VentilatorMY0193, "VentilatorDown")$

The processing of this rule will get the trend with the difference of continuous temperature measuring of indoor and outdoor temperature, and also an instance of the property ("VentilatorDown") *currentMalfunction* of concept *HydraDevice* (which is VentilatorMY0193) will be inferred. Then the Malfunction ontology will be checked for the resolution of the problem based on the malfunction cause. In our case, Malfunction ontology gives us the solution as the "power supply off because of fuse blown".

Usage of Malfunction and Device ontology

For example, Bjarne get a warning of "Grundfos-PumpMQ345 failed to start", which is a high priority task for him as the pump is used for feeding the pigs. A diagnosis task is initiated to check what is wrong with the pump, but as a newly installed pump, there is still no error resolution to this model of pump in the Malfunction ontology. As a further step, the diagnosis system will conduct subsumption reasoning and search for the device *Type* in the Device ontology, which is found as *FluidPump*, and then its manufacturer is also queried. Now another query to the Device ontology will get a similar pump called *GrundfosPumpMQ335* as of the same type from the same manufacturer "Grundfos". And based on the name of the error and pump type, the solution from a query to Malfunction ontology is suggested "replace a capacitor", which is happily the solution to solve the problem.

4.2 Diagnosis manager architecture

Based on the current diagnosis requirements, and also the status of OWL/SWRL, we come up with the following architecture for the Diagnosis Manager as shown in Figure 4, in which the *Component Control* and *Change Management* are enclosed with dashed line, taken Kramer and Magee [6] three Layered architecture as a reference model.

The bottom of the architecture is the ontologies/rules, in which knowledge of devices, and state based diagnosis are encoded. When there are state change events, the device state machine instance in the state machine ontology need to be updated, and also these state changes will be published with state machine state changes as event topic. The Diagnosis Manager is an event subscriber to the state machine state change events, it will then update the corresponding state instances in the ontology. At the same time, this will trigger the diagnosis of the device status, executing the SWRL rules to monitor the health status of devices, and also trigger the reasoning of possible device errors and their resolutions. The Diagnosis Manager will publish the diagnosis results as an event publisher.

The Diagnosis Manager mainly runs on powerful PC or a proxy for an embedded device running on a powerful node. For those node with limited capabilities, only state will be reported, which can delegate its own diagnosis to other node or its proxy.

For the actual implementation, we adopted a mix of the Blackboard architecture style and the Layered architecture, and use the observer pattern in both the updating of state machine ontology and inferred result parsing.

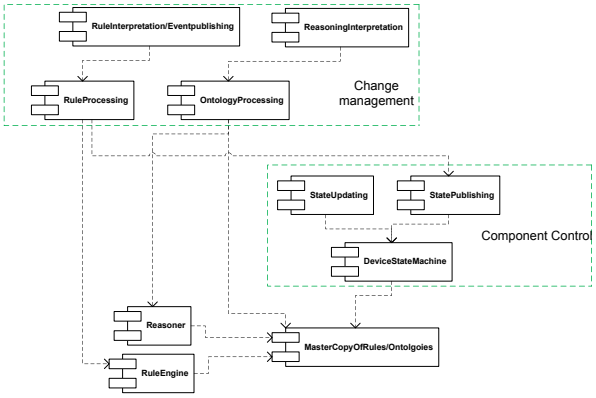

Figure 4. Diagnosis Manager architecture

5 Evaluation

5.1 Extensibility

At present, the extensibility is evaluated by the applicability to new devices added to a system. We started the development of Diagnosis Manager with the rule for temperature monitoring. After finishing the implementation and testing, we then try to handle the flowmeter diagnosis rules. The steps involved are:

1. Add the flowmeter device to the *HydraSystem* concept instance called "Pig" in the Device ontology.

2. Add the flowmeter state machine instance to the StateMachine ontology.

3. Add the flowmeter state machine instance to the has-StateMachine property of the "flowmeter" device.

4. Add flowmeter diagnosis rule to the DeviceRule ontology.

After this, we test the Diagnosis Manager and it runs very well. No single line of Diagnosis Manager code needs to be changed. In summary, the adding of new devices to a certain system is very straightforward. The adding of new devices can be at run time, if the rules for the new devices are existing, then the diagnosis process can be directly working for the new devices.

5.2 Performance

The following software platform is used for measuring performance: Protege 3.4 Build 125, JVM 1.6.02-b06, Heap memory is 266M, Windows Vista. The hardware platform is: Thinkpad T60 Core2Duo 2G CPU, 7200rpm harddisk, 2G DDR2 RAM. The time measurement is in millisecond. The size of DeviceRule ontology is 210,394 bytes, and contains 22 rules.

We measured the performance as shown in Table 1. An interesting thing is after some time of running, the Diagnosis Manager is running stably with the total time in 260-270 ms for processing an event, a bit faster than the one when it starts. Here the parsing of the inferred result is running in a multi-threaded way in the Diagnosis Manager.

Update	InferringTime	AfterEventTillInferred
383	380	382
322	319	321
282	278	282
272	269	271
265	263	265
270	267	269
268	266	269

Table 1. Diagnosis Manager performance

5.3 Scalability

The scalability is evaluated through clients continuously publishing their states (thermometers and flowmeters) as events, in an almost parallel way and each of the client sends messages as fast as possible in a loop. Then we measure how long it will be, starting from the publishing till the end of inferring and publish related inferring result. Time needed (y-axis) is shown in Figure 5 (x-axis shows the number of events) . We can see that the time taken is in linear with the events need to be processed.

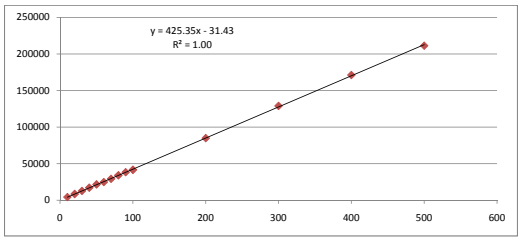

Figure 5. Diagnosis Manager scalability

6 Related work

Kramer and Magee [6] recently proposed a reference model for self-managed systems, which is composed of component control, change management and goal management. In this paper, we largely followed this work for the Layered architecture, but mainly focus the component control and change management. At the same time, a mix of Blackboard architecture and Layered architecture are applied to improve performance and extensibility.

Self-healing is one of the main challenges to autonomic pervasive computing. Generally speaking, our approach applied the same idea of ETS [2], in terms of the using of states for detecting source of failure, and then notification

of failure source. And this process is actually universal for error detections. Our ontology and SWRL rule based approach provides a way of intelligent detection and resolution, which is not easily achievable by ETS.

Work in [1] shares some similarity with us on the usage of semantic web approach for achieving self-managing. Our approach is non-intrusive, SWRL rules are automatically executed using state machine instead of explicitly inserting sensor code to program, and is more suitable for the characteristics of pervasive devices.

Various failures in a pervasive system are classified in [4], and an architecture for fault tolerant pervasive computing is proposed. We focus not only on device failure monitoring using the device state machine, but also system level detection using the relationships of different state machine instances. In addition, our approach can be more intelligent in terms that ontology reasoning can help the diagnosis.

There are many researches from traditional artificial intelligence point of view dealing with the diagnosis in various field, e.g. [3]. These traditional approaches are not utilizing the context ontologies that are already there in pervasive systems and are used for context-awareness and other purposes. The open world assumption in OWL/SWRL and hence in our approach makes our proposed approach well suited for the openness of the pervasive computing environment, which automatically rejects the approaches using Prolog kind of rules which use close world assumption.

7 Conclusions and future work

OWL/SWRL is adopting an open world assumption which is in nature very suitable for the pervasive computing systems, where the openness and dynamicity dominate the interaction and function. OWL is widely used in pervasive computing, for the purpose of context awareness, service selection and composition. The potentials of OWL and context awareness could be further extended as we have shown in this paper.

Diagnosis is the most important step for achieving self-healing, which is a challenge in pervasive computing. We present a semantic and state machine based diagnosis approach using OWL ontology and SWRL, for the Hydra middleware. The malfunction information and its resolution are encoded in an OWL ontology, and can be used at run time to infer the solution to the malfunction, and further to fulfill self-healing activities. SWRL is used to develop monitoring and diagnosis rules, which can help make intelligent decisions when there is malfunction occurs. IPSniffer will help diagnosis on devices that are dead or no response which provides fault tolerance in our approach.

The evaluations relieved us for the worrying of performance of the OWL/SWRL based Diagnosis Manager. In order to improve performance, we followed a mix of both the Blackboard architecture style and the Layered architecture style. The evaluations show that the Diagnosis Manager is usable in terms of extensibility, performance and scalability. The proposed approach provides an uniform, coherent and natural way to fully utilize the existing OWL/SWRL reasoning power, and extend it for considering the dynamic aspects of the pervasive system for diagnosis, which is very suitable for the characteristics of the pervasive computing environment.

We are improving the IPSniffer based diagnosis that only reports invocation relationships at present. The integration with security manager and ontology manager are under way. Probability in OWL/SWRL is to be added in the future to make the diagnosis more intelligent. More experiments in a larger scale will be conducted for testing the resolving of rule conflicts, accuracy of diagnosis and so on.

Acknowledgements

The research reported in this paper has been supported by the Hydra EU project (IST-2005-034891).

References

[1] B. J. O. A. R. Haydarlou, M. A. Oey and F. M. T. Brazier. Use-case driven approach to self-monitoring in autonomic systems. *The Third International Conference on Autonomic and Autonomous Systems*, 2007.

[2] S. Ahmed, M. Sharmin, and S. Ahamed. ETS (Efficient, Transparent, and Secured) Self-healing Service for Pervasive Computing Applications. *International Journal of Network Security*, 4(3):271–281, 2007.

[3] R. Barco, L. Díez, V. Wille, and P. Lázaro. Automatic diagnosis of mobile communication networks under imprecise parameters. *Expert Systems With Applications*, 2007.

[4] S. Chetan, A. Ranganathan, and R. Campbell. Towards fault tolerant pervasive computing. *Technology and Society Magazine, IEEE*, 24(1):38–44, 2005.

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

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

[7] I. A. Project. Amigo middleware core: Prototype implementation and documentation, deliverable 3.2. In *IST-2004-004182*, 2006.