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) 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

1OWL homepage. http://www.w3.org/2004/OWL/
2SWRL specification homepage. http://www.w3.org/Submission/SWRL/
3http://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 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 state machines. The relationship between Service and Port is akin to the concepts described in the WSDL 1.1 specification5. The state machines 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

---

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

5http://www.w3.org/TR/wsdl
Table 1. An Example Transition Notification

<table>
<thead>
<tr>
<th>Topic</th>
<th>Statemachine/transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>Value</td>
</tr>
<tr>
<td>DeviceId</td>
<td>pico_th03_0xA12A</td>
</tr>
<tr>
<td>DeviceType</td>
<td>PicoTh03</td>
</tr>
<tr>
<td>FromStateName</td>
<td>Measuring</td>
</tr>
<tr>
<td>ToStateName</td>
<td>Measuring</td>
</tr>
<tr>
<td>EventName</td>
<td>measured</td>
</tr>
<tr>
<td>Result</td>
<td>17.54</td>
</tr>
</tbody>
</table>

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 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 3. Flamenco Probe Deployment](image1)

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:

<table>
<thead>
<tr>
<th>Approach/criteria</th>
<th>dc</th>
<th>pv</th>
<th>qua</th>
<th>inc</th>
<th>for</th>
<th>app</th>
<th>re</th>
<th>du</th>
<th>rca</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key-value</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>++</td>
<td>+</td>
<td>-</td>
<td>++</td>
</tr>
<tr>
<td>Markup</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Graphic</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Object orientation</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Logic based</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Ontology based</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>++</td>
</tr>
</tbody>
</table>

Table 2. Context model comparisons

![Image of Figure 5](image-url)

**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](image)

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 has-Device 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 ontology⁹. 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

---

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 Aml 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\footnote{LeJOS homepage. http://lejos.sourceforge.NET/} 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.

\[ \text{JavaVM} \equiv \exists \text{runs.JavaByteCode}, \text{SuperWaba} \subseteq \text{Library} \]

\[ \text{Then} \quad \text{SuperWaba} \subseteq \text{JavaVM} \]

The usage of transitive property can also derive useful new information, for example requiresMoreMemory is transitive, we have the following axioms:

\[ \text{requiresMoreMemory(CDC CLDC)}, \]

\[ \text{requiresMoreMemory(J2SE CDC)} \]

\[ \text{Then} \quad \text{requiresMoreMemory(J2SE CLDC)} \]

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:

\[ \text{MobilePhone(?device)} \land \]

\[ \text{hasHardware(?device, ?hardware)} \land \]

\[ \text{primaryBattery(?hardware, ?battery)} \land \]

\[ \text{batteryLevel(?battery, ?level)} \land \]

\[ \text{swrlb: lessThanOrEqual(?level, 0.1)} \]

\[ \rightarrow \text{VeryLowBattery(?device)} \]

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...
rules, the symbol “∧” means conjunction, and “?x” stands for a variable, “→” 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.

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 FlamencoProbe calling rule

---

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

<table>
<thead>
<tr>
<th>Update</th>
<th>InferringTime</th>
<th>AfterEventTillInferred</th>
</tr>
</thead>
<tbody>
<tr>
<td>843</td>
<td>843</td>
<td>843</td>
</tr>
<tr>
<td>906</td>
<td>906</td>
<td>906</td>
</tr>
<tr>
<td>922</td>
<td>922</td>
<td></td>
</tr>
<tr>
<td>953</td>
<td>938</td>
<td></td>
</tr>
</tbody>
</table>

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 maximum improvement of 52%, and a maximum of 69%.

<table>
<thead>
<tr>
<th>Update</th>
<th>Inferring Time</th>
<th>AfterEventTillInterred</th>
</tr>
</thead>
<tbody>
<tr>
<td>328</td>
<td>328</td>
<td>328</td>
</tr>
<tr>
<td>297</td>
<td>281</td>
<td>297</td>
</tr>
<tr>
<td>297</td>
<td>281</td>
<td>297</td>
</tr>
<tr>
<td>344</td>
<td>344</td>
<td>344</td>
</tr>
</tbody>
</table>

**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 [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


