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.

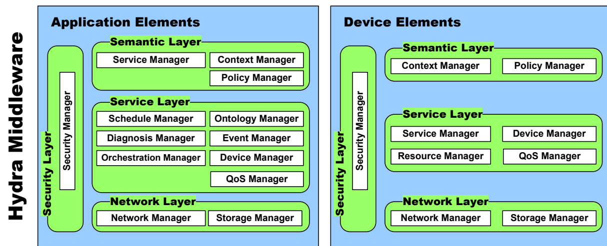

Figure 1. Hydra middleware Layered architecture

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.

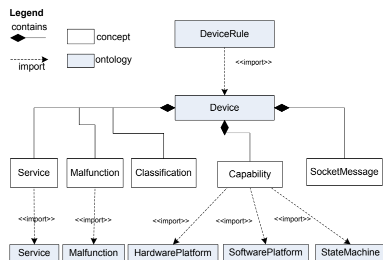

Figure 2. Diagnosis ontologies structure

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.

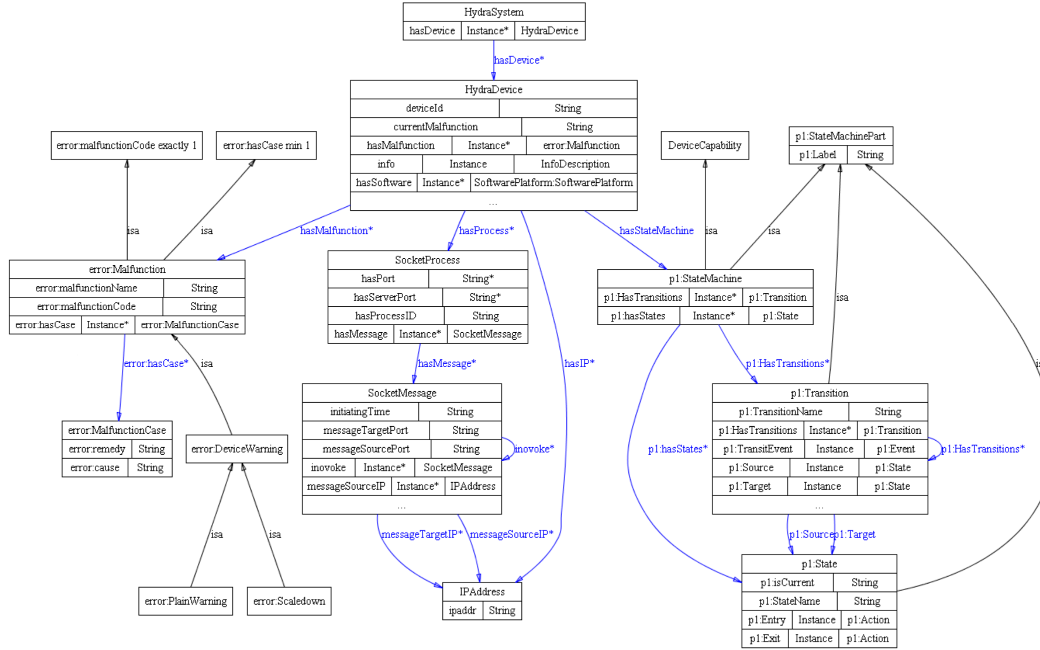

Figure 3. Partial details of the Diagnosis Manager used ontologies

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.

$device : hasStateMachine(device : PicoTh03_Outdoor, ?sm)$
$\wedge\ statemachine : hasStates(?sm, ?state) \wedge$
$statemachine : doActivity(?state, ?action) \wedge$
$statemachine : actionResult(?action, ?r) \wedge$
$statemachine : historicalResult1(?action, ?r1) \wedge$
$statemachine : historicalResult2(?action, ?r2) \wedge$
$statemachine : historicalResult3(?action, ?r3) \wedge$
$swrlb : add(?tempaverage, ?r1, ?r2, ?r3) \wedge$
$swrlb : divide(?average, ?tempaverage, 3) \wedge$
$swrlb : subtract(?temp1, ?r, ?r1) \wedge$
$swrlb : subtract(?temp2, ?r1, ?r2) \wedge$
$swrlb : subtract(?temp3, ?r2, ?r3) \wedge$
$swrlb : add(?temp, ?temp1, ?temp2, ?temp3) \wedge$
$swrlb : greaterThan(?average, 12.0) \wedge$
$swrlb : lessThan(?average, 33.0) \wedge$
$swrlb : lessThan(?temp, 0) \wedge$
$device : hasStateMachine(device : PicoTh03_Indoor, ?sm_b)$
$\wedge\ statemachine : hasStates(?sm_b, ?state_b) \wedge$
$statemachine : doActivity(?state_b, ?action_b) \wedge$
$statemachine : actionResult(?action_b, ?r_b) \wedge$
$statemachine : historicalResult1(?action_b, ?r1_b) \wedge$
$statemachine : historicalResult2(?action_b, ?r2_b) \wedge$
$statemachine : historicalResult3(?action_b, ?r3_b) \wedge$
$swrlb : subtract(?temp1_b, ?r_b, ?r1_b) \wedge$
$swrlb : subtract(?temp2_b, ?r1_b, ?r2_b) \wedge$
$swrlb : subtract(?temp3_b, ?r2_b, ?r3_b) \wedge$
$swrlb : add(?temp_b, ?temp1_b, ?temp2_b, ?temp3_b) \wedge$
$swrlb : greaterThan(?temp_b, 0) \rightarrow device : currentMalfunction(device : VentilatorMY0193, "VentilatorDown")$

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 "GrundfosPumpMQ345 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.

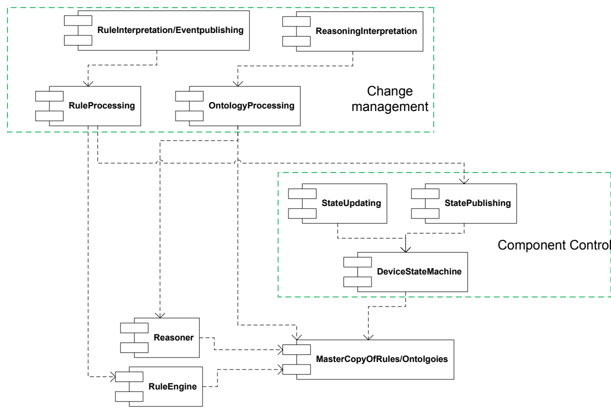

Figure 4. Diagnosis Manager architecture

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 hasStateMachine 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.

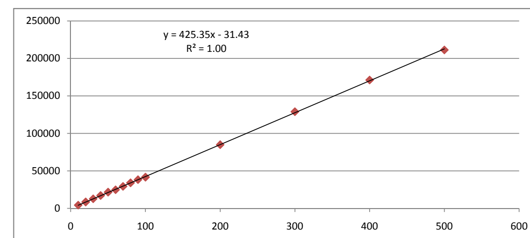

Figure 5. Diagnosis Manager scalability

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.