Towards a Flexible Middleware for Context-aware Pervasive and Wearable Systems

Marco Muro, Michele Amoretti, Francesco Zanichelli, Gianni Conte

Distributed Systems Group, Department of Information Engineering, Università degli Studi di Parma, Parco Area delle Scienze 181/A, Parma, Italy

Corresponding Author:
Michele Amoretti,
e-mail: michele.amoretti@unipr.it
tel: +39 0521 906147
fax: +39 0521 905723

The total number of words of the manuscript, including entire text from title page to figure legends: 5379
The number of words of the abstract: 131
The number of figures: 8
The number of tables: 0
Ambient intelligence and wearable computing call for innovative hardware and software technologies, including a highly capable, flexible and efficient middleware, allowing for the reuse of existing pervasive applications when developing new ones. In the considered application domain, middleware should also support self-management, interoperability among different platforms, efficient communications, and context-awareness. In the on-going “everything is networked” scenario scalability appears as a very important issue, for which the peer-to-peer (P2P) paradigm emerges as an appealing solution for connecting software components in an overlay network, allowing for efficient and balanced data distribution mechanisms. In this paper we illustrate how all these concepts can be placed into a theoretical tool, called Networked Autonomic Machine (NAM), implemented into a NAM-based middleware, and evaluated against practical problems of pervasive computing.

Middleware, context-awareness, autonomic, peer-to-peer

Glossary of terms when relevant
Introduction

The concept of ambient intelligence (AmI) refers to environments that are aware and responsive to the presence of people, proactively supporting them in their daily lives. AmI overlaps with other ICT research topics, such as pervasive computing, networked embedded systems and artificial intelligence. Very near to AmI, wearable computing hopes to produce a paradigm shift of how a computer should be used. A person’s computer should be worn, much as eyeglasses or clothing are worn, and interact with the user based on the situational context. The latter may be defined as “any information that can be used to characterize the situation of an entity” [Yu06]. An entity is a person, place or object that is considered relevant to the interaction between the user and the application, including the user and the application themselves. Examples of context information include a user’s location or profile, the time, local resources of the mobile device and/or related sensors, available services, etc.

The development of AmI and wearable computing software is a complex and error-prone task because it must cope with heterogeneous service infrastructures and with system dynamics in an open network [Fuentes09]. It is therefore necessary to adopt a highly flexible and efficient middleware to integrate existing services with new ones, and also to support efficient communication (also in mobility) and adaptation of applications to the current context [Jaroucheh09], [Narayanaswami06], [Song07]. Middlewares proposed in the literature differ significantly in terms of features, in particular regarding the inclusion of reasoning capabilities.

On the other hand, scalability is considered unanimously a very important issue, in the ongoing “everything is networked” scenario. The peer-to-peer (P2P) paradigm emerges as an appealing solution, allowing for efficient and balanced data distribution among connected software components - which are crucial in pervasive applications, being them mostly based on the publish-subscribe paradigm, with hundreds of potential context producers. A P2P system is intrinsically complex, being composed of several interconnected parts (the peers) that play both client and server roles, and as a whole may exhibit one or more properties (i.e. behavior) which are not easily inferred from the properties of the individual parts [Ottino04]. This apparent weakness may turn into a strong point, if peers are
provided with autonomic mechanisms for detecting, diagnosing and repairing fail-
ures, and adapt their behaviors to changes in the environment - which is a highly
appealing feature for a wide range of pervasive systems. We argue that, by in-
creasing the context-awareness of monitoring data exchanged within and among
autonomic peers, it is possible to efficiently sense network conditions as well as
the level of provided services and perform corrective actions.

In this paper we illustrate how all these concepts can be placed into a theoretical
tool, called Networked Autonomic Machine (NAM), implemented into a NAM-
based middleware, and evaluated against practical problems of pervasive comput-
ing.

State of the Art

Regarding middleware for AmI and wearable systems, the literature provides a
wealth of approaches, for which it is not possible to cover them all here. We limit
the discussion to a couple of important and recent european projects, namely HY-
DRA and PERSONA, as well as four research works that take into account auto-
nomicity and code refactoring, besides the common idea of an extensible middle-
ware supporting service discovery, distributed information sharing, context man-
agement.

The HYDRA project [Eisenhauer09] develops middleware for networked embed-
ded systems that allows developers to create ambient intelligence applications
utilizing device and sensor networks. The proposed middleware provides support
for embedding services in devices and for proxying services for devices.
Moreover, middleware supports dynamic reconfiguration and self-configuration.
The dynamic context is modeled with runtime concepts and properties in four on-
tologies defined within the HYDRA project: StateMachine, Malfunction, QoS,
and Device. The Semantic Web Rule Language (SWRL) is used to define com-
plex context events, such as ”the user is far away from home”, that can be used to
take actions, like ”surveillance system is switched to the highest security level”.

Another remarkable solution is that implemented within project PERSONA
[FidesValero08], consisting in an architecture based on a set of different commu-
nication buses, each of them adopting specific and open communication
strategies. Its design has been driven by the need for a self-organizing infrastruc-
ture allowing the extensibility of component/device ensembles in an ad hoc fash-
Towards a Flexible Middleware

In order to achieve this goal, the PERSONA Middleware implements distributed coordination strategies that provide the necessary service discovery, service orchestration and service adaptation functionalities. Components linked with the PERSONA Middleware may register with some of these communication buses, find each others and collaborate through the local instances of the buses. Input and output buses support multi-modal user interactions with the system. The context bus is an event-based channel to which context sources are attached. Published events may be re-elaborated and transformed in high level events (situations) by components that have subscribed to the bus (e.g. context reasoners). The service bus is used to group all the available services, being them atomic or composite (whose availability is managed by a Service Orchestrator component). Last but not least, the PERSONA middleware supports different network protocols, allowing to integrate almost any kind of sensor network.

More specifically dedicated to ubiquitous sensor networks, the LAMSES middleware [Jeong2010] can process large-scale sensor data and provide context-aware functionalities. With respect to PERSONA, LAMSES is much more “heavy” because it does include five components: service management, meta-information management, sensing management, state management, and control and query management. In particular, the sensing management component receives sensed information from sensor nodes, integrates sensed data, analyzes data through a context-aware recognizer, and manages specific events.

Based on a similar approach - for which the middleware is not only a connection infrastructure, but also as a service provider - the LAICA project exploits the agent paradigm to model devices and sensors, and develops an ad-hoc middleware to orchestrate all the involved components [Cabri08]. To this purpose, the middleware can accept two kinds of incoming data: raw data coming from sensors and commands issued by components to obtain a service, for example to activate a sensor. Another detail about the middleware is that it can exploit mobile agents to face unordinary situations. For example, if an agent (either sensor or effector) is no more responding to the middleware commands, the middleware can send a mobile agent to the device the former is attached to, in order to evaluate and try to locally correct the situation.

Regarding node refactoring, the SelfLet approach proposed by Devescovi et al. [DeVescovi07] is very appealing. It does target autonomic systems whose nodes
(i.e. the SelfLets) may change their structure at run time in order to acquire new capabilities (by dynamically loading software components that implement specific abilities), or to lose their capabilities (by undeploying components). Each SelfLet has a basic behavior that implements the SelfLet life cycle, specified as a set actions to perform, by a set of Goals to use or achieve, and by a set of autonomic rules supporting application dependent self-configuration. Each SelfLet has a knowledge base, for which adaptation is epigenetic, i.e. based on learning and knowledge transmission. Conversely, we are currently working on the problem of adaptive re-structuring of peers using a phylogenetic approach, i.e. memoryless transformations, with the purpose of reducing overhead and achieving real-time self-adaptation. We are also investigating hybrid approaches.

Another interesting approach for peer restructuring has been proposed by Tyson et al. [Tyson08], that show how survival of the fittest has been implemented into the Juno middleware. On receipt of a superior component, Juno dynamically reconfigures the internal architecture of the peer, by replacing the existing component with the new one. For example, a node would evolve its maintenance algorithms by obtaining a new maintenance component that offers the correct interface. The old component would then be removed from the software architecture and replaced by the new one. All subsequent maintenance functionality would then be performed by the new component. To support this, interchangeable components must offer identical interfaces. Moreover, to evaluate a new component, all meta-values are defined relative to the default component that defines a base-line for all equivalent components. This approach removes the necessity for other components and applications to possess semantic knowledge of quantitative values. Instead, it is possible to simply consider their capabilities as relative to each other.

To summarize, middleware for AmI-oriented and wearable computing can be either lightweight (like PERSONA) or heavyweight (like LAMSES or LAICA), depending on how many “reasoning” features it includes. In our opinion, resource-demanding functionalities like context storage, context reasoning, message routing, etc. should not be provided by the middleware. Instead, they should be implemented as “pluggable components” that peers may deploy or not, depending on their hardware resources. The middleware should provide mechanisms for plugging and unplugging these components at runtime. As illustrated in the following sections, our lightweight middleware solution copes well with such re-
quirements, and advances over the state of the art by introducing the enforcement of autonomic principles (through self-management policies) within software components. The advantages of this approach are manyfold, namely reduction of maintenance issues, self-adaptation of the system to changing conditions in the environment, as well as an improved quality of service.

**NAM framework**

The middleware design approach we propose is based on the Networked Autonomic Machine (NAM), a theoretical tool which describes a computing element that

- hosts functional modules provided with goals, functional policies and self-management policies;
- dynamically deploys/undeploys functional modules and services;
- is connected to a set of other NAMs.

In a NAM-based distributed system, the achievement of global self-governance (autonomy) and self-management (autonomicity) depends on the policies adopted at the functional level. For example, by providing all NAMs with a peer-to-peer overlay management module, it may be possible to enable their cooperation in routing messages to discover and use/download new services and functional modules.

Formally, a NAM is a tuple $\langle R;F;L \rangle$ where $R$ is a set of physical resources (e.g. processing units, storage devices, download/upload bandwidth), $F$ is a set of functional modules and $L$ is a set of links to other NAMs.

The definition of functional module is:

$$f = \langle \text{C}_{\text{in}};\text{C}_{\text{out}};\text{OBJ};\text{FP};\text{SMP}; \text{S}_{f};\text{S}_{r} \rangle$$

where

- $\text{C}_{\text{in}}$ is the set of consumed context events;
- $\text{C}_{\text{out}}$ is the set of provided context events;
- $\text{OBJ}$ is the set of objectives;
- $\text{FP}$ is the set of functional policies;
- $\text{SMP}$ is the set of self-management policies according to which the functional module performs self-configuration, self-healing, self-optimization and self-protection;
- $\text{S}_{f}$ is the set of provided services;
• $S_f$ is the set of services the functional module can consume.

Context events are information pieces that describe changes in the environment. From a grammatical point of view, a context event may be a RDF simple fact (i.e. a triple $\langle$ subject;predicate;object $\rangle$ [Klyne04]), or something more complex. Context awareness is usually implemented following the publish / subscribe paradigm, for which context consumers express interest in one or more context event classes, and only receive context events that are of interest, without knowledge of what, if any, context providers there are. Context events are published by context providers periodically or sporadically - never on-demand. On the other hand, a service provider is a software entity that exposes a set of services to service consumers, and execute such services upon request.

Functional policies are rules, algorithms, evolutionary plans, etc. the functional module adopts in order to meet its objectives. When receiving one or more context events in $C_{in}$, the functional module may react by publishing a set of context events in $C_{out}$, by executing a set of services in $S_f$, or by calling a set of services in $S_r$.

A service consists in a unit of work executed by a service provider to achieve the results desired by a service consumer. In general, a service is a tuple $s = \langle I;O;P;E \rangle$

where $I$ is a set of input parameters, and $O$ is a set of output parameters. Each I/O parameter has a type, i.e. a class (still using the ontological terminology). It is important that service consumers and service providers share the same domain ontologies in order to have a common understanding of shared services. Semantic descriptions of services are used to organize service advertisements in centralized or distributed repositories, allowing to efficiently retrieve and use services in the network. $P$ and $E$ are the precondition and effect sets, respectively. Such optional parameters are expressed in the form of logical conditions which can assume the true or false value. Preconditions must be verified in order to invoke the service, while an execution effect may become a precondition for the successive invocation in a composition scenario. The IOPE approach is adopted for example in the OWL-based Web service ontology called OWL-S [12].

An atomic service is defined as the minimal executable function unit, that cannot be decomposed and whose execution can transform a given state to another state. It is represented as a tuple
as = \langle I;O;P;E;Q \rangle

where \( Q \) is the set of exposed quality of service (QoS) properties, such as reliability, availability, performance, security and timing. Each node can provide different atomic services. The number of concurrent service instances and the quality of service (QoS) of each instance at a certain time depends on the current availability of hardware resources on the node. Intra- or inter- NAM services can be statically or dynamically aggregated (proactively or on-demand) to realize new complex tasks. A composite service is a tuple

\[ cs = \langle I;O;P;E;Q;G_w \rangle \]

where \( G_w \) is the rule that allows to combine atomic services; this rule is represented as a directed workflow graph

\[ G_w = \langle S_w; L_w \rangle \]

where \( S_w \) is a set of services (both atomic and composite) and \( L_w \) is a set of links that represent transitions (i.e. I-O connections) among services. Figure 1 shows the internal structure of a NAM, with resources, functional modules, and services.

Fig. 1. A network of NAMs. Functional modules use resources and provide atomic services (as), which can be aggregated and orchestrated as composite services (cs) by any interested functional module.

NAMs are able to dynamically reconfigure their structure, by enabling new functional modules or services, or disabling those that are no more necessary.

Autonomic reasoning

As illustrated in figure 2, functional modules are able to perceive (i.e. being context-aware), decide and act. Like agents, functional modules have objectives and
functional policies for achieving their objectives. Unlike agents, functional modules are also provided with highly specific self-management policies.

Decision is the (policy-based) ability to infer new context events, service calls, or output (to some users, or to a service request). The decision process derives new facts (output) from known facts (input). Policies can be statically defined or dynamically learnt.

Facts and rules should be defined according to ontologies shared among NAMs. The ontological approach tries to reduce the gap between informal language and programming language by means of formal ontology languages. Formal ontology languages have a precise semantics that allow for unambiguous specifications of domain knowledge. These languages are easy enough to learn quickly (much faster than programming languages), not just by software engineers, but also by domain experts.

There is a variety of languages which can be used for representation of conceptual models, with varying characteristics in terms of their expressiveness, ease of use and computational complexity [Stevens00]. Normann and Putz suggest to use Prolog to define ontologies for ambient intelligence systems [Normann10]. We argue that Prolog may be applied to any autonomic distributed system, once provided with mechanisms for checking the consistency of fact and rules. Indeed, two main issues must be considered. First, a recently consumed context event (fact) may supersede a previous one, or not. For example ”Jack is sitting”, received after ”Jack is walking”, should replace the latter in the knowledge base of the NAM. On the
other side, "Jack is talking” may be considered as supporting information for "Jack is walking”. Second, the set of rules may change over time, taking care that new rules do not conflict with existing ones.

Interestingly, it has been proven that logic programming can be used for defining rules that produce new knowledge from an existing OWL knowledge base. In particular, it is possible to map OWL concepts to Prolog facts, thus allowing to infer new facts from existing ones, by means of Prolog rules [Obrst07]. This is very important, because the ontology Web language OWL has been established as a standardized and widely accepted means for machine-to-machine representation and exchange of knowledge [OWL09] - for example, OWL is a basic building block of the Semantic Web (also called Web 3.0).

In conclusion, we propose the use of Prolog to allow experts to define domain theories, sensors to provide context events as facts, and functional modules to perform reasoning over the resulting knowledge base. Such an outcome can be exported to the Web by means of OWL, a widely adopted, machine-readable language. This vision is illustrated in figure 3.

![Figure 3](image)

Fig. 3. Theories (defined by experts) and facts (generated by users and the environment), collected and shared by NAMs, contribute to the creation of a large, dynamic knowledge base that may aid experts in the refinement of theories, and benefit users by improving their quality of life.
Nam4j middleware

As previously stated, the PERSONA middleware [FidesValero08] is a good example of lightweight solution for integrating software components in AmI scenarios. Our experience within the PERSONA project [Amoretti09] is the basis of the Nam4j open project, that we recently started with the purpose of developing a portable implementation of the NAM framework (http://code.google.com/p/nam4j/).

The Nam4j middleware provides interfaces and abstract classes for creating functional modules and NAMs, using a modular approach (figure 4).

![Inheritance relationships between interfaces, abstract classes and concrete classes in the Nam4j core API. Interfaces define basic methods that are common to all functional modules. Abstract classes implements such basic methods. Concrete classes define and implement application-specific methods.](image)

Each functional module may contain several instances of Context Provider (CP), Context Consumer (CC), Service Provider (SP), Service Consumer (SC), as illustrated by the class diagram in figure 5. This approach strongly supports code reuse, since CP, CC, SP and SC classes, once implemented, can be included in several different functional modules.
Towards a Flexible Middleware

Fig. 5. Class diagram of a generic functional module.

**Functional module example: \texttt{f}^{\text{over}}**

A first example of functional module we have developed, called \texttt{f}^{\text{over}}, provides publish and lookup services based on the Chord protocol [Stoica01], implemented using Sip2Peer (http://code.google.com/p/sip2peer/). The latter is an open-source, multi-platform SIP-based middleware that allows to develop any peer-to-peer scheme, overcoming typical issues such as Network Address Translation (NAT) traversal.

To implement \texttt{f}^{\text{over}} we have chosen Chord because of its high performance in routing messages, which strongly supports the realization of efficient publish/subscribe mechanisms [Baldoni05].

Other than publish and lookup services, \texttt{f}^{\text{over}} is provided with self-management policies for autonomic configuration, connection to the network, and preservation of network consistence.

**Functional module example: \texttt{f}^{\text{act}}**

Another functional module we have implemented, called \texttt{f}^{\text{act}}, is a context reasoner provided with activity monitoring policies. Here we have used Prolog to define formal concepts of the ontology, as well as rules (i.e. the theory). Reasoning is based on tuProlog [Denti05], which is a Java-based lightweight Prolog engine that provides a straightforward API to implement simple or more complex Prolog program within Java code, or to read existing Prolog expressions from a file or from a
database. Once one or more Prolog theories have been acquired, it is possible to use them to evaluate facts and derive new facts. Examples of theories managed by \( f_{\text{act}} \) are illustrated in the remainder of the paper.

Validation examples

In this section we illustrate three sample scenarios in the context of e-health and AmI where the nam4j middleware is fruitfully applied. As general testbed we consider a network of six NAM instances, each one carrying a specific functional module:

- \( f_{\text{wear}} \) - used by NAM-1 with its generic wearable sensors, to generate context events related to user parameters of interest (biomedical, physiological and activity data, such as electrocardiograms, electromyographs, body temperature and blood pressure, electro-dermal activity, heartbeat rate, movement or muscular activity);
- \( f_{\text{env}} \) - used by NAM-2 with its generic environment sensors, to generate context events related to the state of the environment in which the user lives;
- \( f_{\text{cam}} \) - used by NAM-3 that handles a camera and generates context events describing the posture of the user;
- \( f_{\text{act}} \) - used by NAM-4 as to reason (using Prolog) about context events generated by NAM-1, NAM-2, NAM-3 and - if necessary - invokes services placed on NAM-5 and NAM-6;
- \( f_{\text{call}} \) - used by NAM-5 to provide a service for sharing information with experts or for emergency calls.
- \( f_{\text{user}} \) - used by NAM-6 to collect user input from his/her mobile device.

NAM-1 and NAM-2 host several instances of \( f_{\text{wear}} \) and \( f_{\text{env}} \), respectively, that can be deployed or undeployed at runtime by the middleware depending on which sensors are actually worn by the user, or set up in the environment.

Every NAM is equipped with the overlay module \( f_{\text{over}} \), described in the previous section, to discover nodes in the local network and communicate with them.

Scenario 1: Monitoring and Data Collection

Let us consider a patient affected by a chronic disease (e.g. heart pathologies), that needs periodic monitoring of her/his condition. Such a patient is provided with wearable sensors which are designed to collect biomedical, physiological and
activity data. To this purpose, it is necessary to configure functional modules which collect different types of data in according to patient needs. Figure 6 describes a NAM-based solution for such a patient monitoring scenario.

![Diagram of NAM-based solution](image)

Fig. 6. NAM-based solution for the patient monitoring scenario.

The logic that guides \( f_{act} \) in its choices, defined using Prolog, is the following:

\[
\text{isNewData}(X) :\overline{\text{measuredSystolicPressure}}(X,Y) ; \quad \text{measuredDiastolicPressure}(X,Y).
\]

That is, for a generic user \( X \) isNewData is true only if there is a new value for measured pressure from the sensor. Wearable sensors periodically produce information about systolic and diastolic pressure, which are sent to a mobile device that runs NAM-1; its functional modules generate two different context events, such as:

\[
\text{measuredSystolicPressure}(\text{user}, 69), \quad \text{measuredDiastolicPressure}(\text{user}, 109).
\]

NAM-4 catches context events produced by NAM-1, and stores them in the knowledge base. If new facts differ to a significant clinical extent from previous ones, NAM-4’s \( f_{act} \) calls a service provided by \( f_{call} \) (hosted by NAM-5) that shows patient-related information to the doctor.
Scenario 2: Wandering Detection

In this case we consider demented or disoriented patients living at home with parents or in nursing homes. Such patients often wander at night. A bracelet can generate an information about the current position of the patient in the house, while a carpet placed in front of the bedroom can give the information about the movements of the patient. Furthermore there are other sensors in each window and in the entrance to determine if these are open or closed. Bracelet data are collected by NAM-1, that periodically generates a context event; NAM-2 produces context events only if something is changing in the environment, like the opening of a door.

\[
\text{currentPosition(user, hall).} \\
\text{changedStatus(hall, open).}
\]

NAM-4 catches all these context events in order to apply the defined policies and in case to invoke a service call to inform the nurse of the situation.

\[-wanderingDetect(user).\]

The following theory guides the engine in its choices:

\[
\text{wanderingDetect(X)} \leftarrow \text{currentPosition(X, Y),} \\
\text{accessStatus(Y).} \\
\text{accessStatus(X)} : - \text{changedStatus(X, Y), isOpen(Y).} \\
\text{isOpen(A)} : - \text{A = open.}
\]

That is, wanderingDetect for a generic user X is true only if the position of X is place Y and Y is a place in which window or door have been recently open.
Scenario 3: Emergency Situation

In the last scenario we consider a patient affected by diabetes, that may be exposed to hyperglycemia for not taking insulin or to cases of hypo for the opposite situation.

A glucose sensor and a camera are used to periodically analyze the patient’s glucose level and the environment. If a change of the state in the measured variables is detected, the NAMs generate specific context events. NAM-4 listens for such context events: if a user has glucose level out of range and she/he is lying down, NAM-1 and NAM-3 generate context events that are caught by NAM-4, whose functional module $F_{AM}$ detects the danger and provides the patient an alert on some monitor in the house or in the patient mobile device; in the meantime an emergency call is started to parents or directly to first aid. We assume the availability of an intelligent camera able to recognize three different postures for the user (laying down, sitting and standing) and the wearable sensor monitoring in a periodic way the level of glucose in the blood.

For this scenario we can define a theory based on five predicates:

\[
\begin{align*}
\text{emergencyCall}(X) & : \text{laying}(X),
\text{lowGlucoseLevel}(X) \lor \text{highGlucoseLevel}(X).
\text{lowGlucoseLevel}(X) & : \text{measuredGlucose}(X,Y),
\text{limitLowGl}(Y).
\text{highGlucoseLevel}(X) & : \text{measuredGlucose}(X,Y),
\text{limitHighGl}(Y).
\end{align*}
\]
limitLowGl(A) :- A < 60.
limitHighGl(B) :- B > 160.

Fig. 8. NAM-based solution for the emergency scenario.

That is, emergencyCall for a generic user X is true only if X is laying and lowGlucoseLevel or highGlucoseLevel is true, where lowGlucoseLevel for X is true only if the measured glucose level in the blood exceeds the limitLowGl definition (the same applies for highGlucoseLevel).

NAM-1 and NAM-3 generate context events like:

laying(user).
measuredGlucose(user,57).

When NAM-4 catches such context events, it performs the following Prolog query:

?- emergencyCall(user).

If the engine gives a positive answer, NAM-4 provides to produce the user input by invoking the service provided by NAM-6 and to start an emergency call invoking the service exposed by NAM-5. Of course, it is possible to define more complex conditions for triggering an emergency call (e.g. evaluating the agitation, the pulse of the patient and so on), as well as policies for dealing with node failures.
Discussion

In the paper we have analyzed the state of the art of middleware for AmI-oriented and wearable computing, and we have illustrated the formal concept of Networked Autonomic Machine (NAM). We have also illustrated how NAM is being concretized in the Nam4j middleware, which is based on a lightweight approach - i.e. reasoning, communication and other application-specific features are placed outside the middleware, in functional modules. To support this design solution, we have proposed three application scenarios we are addressing with Nam4j.

As future work, we are going to complete the development of Nam4j along with a number of significant demos related to the AmI application domain. In particular, we are focusing on the challenging problem of creating dynamic knowledge bases for NAMs, taking into account explicitly the spatial and temporal dimensions of policies. For example, alarm service execution could depend on user location and time of the day (if the user is watching TV, the alarm could be a visual message on the TV screen; if the user is outdoor, the alarm may be an audio message on his/her smartphone). Moreover, we are working on mechanisms for enforcing self-management (autonomic) policies, to support functional ones.

References


