Context-Awareness in Multi-Agent Systems for Ambient Intelligence[*]

Andrei Olaru

Abstract There is a large body of research that lies at the intersection of the domains of context-awareness, multi-agent systems (MAS) and Ambient Intelligence (AmI) / Ubiquitous Computing (UbiComp). This is because, while multi-agent systems are an appropriate architecture for AmI implementations, one essential requirement for AmI is to be aware of the user's context and to act accordingly. In order to implement context-awareness in a MAS for AmI applications, one must on the one hand choose an appropriate representation for context, that is suitable for agents of all sizes and functions, and, on the other hand, create an agent-based architecture that facilitates communication between agents that share context.
This chapter presents a model, mechanisms and methods for integrating context-awareness in multi-agent systems for AmI. The model is based on experience with several implementations of MAS dealing with various aspects of context-awareness.

1 Introduction

Ambient Intelligence – or AmI – is a pervasive electronic environment that will assist people in their daily lives, in a pro-active but non-intrusive manner [1, 6]. In order to be able to take the appropriate action at the right time, AmI must consider the situation – or context – of the user, in order to help the user, potentially by means of proactive action, without disrupting the user's focus. Context-awareness is therefore a central element in Ambient Intelligence, being instrumental in AmI

[*] The final publication is available at
`http://link.springer.com/chapter/10.1007/978-1-4939-1887-4_33`.

Andrei Olaru
Department of Computers, Faculty of Automatic Control and Computers,
University "Politehnica" of Bucharest, 313 Splaiul Independentei, 060042 Bucharest, Romania,
e-mail: cs@andreiolaru.ro

appearing as "intelligent". An AmI system must "understand" the context of the user before acting upon it.

In context-aware AmI applications, context is many times viewed as set of pieces of information that originate outside the AmI system, and that the system can perceive and manage in order to provide it to the various algorithms involved in decision. However, only so much context information will come from the outside of the system. Applications in the system will also produce information that is relevant to other applications, essentially creating more complex elements of context. We therefore view applications and agents in an Ambient Intelligence system as both consuming and producing context information.

Moreover, there are a few processes related to context information that can be found in most of context-aware applications: the application must be able to detect the context information that is relevant to it; it must detect the appropriate action to take, considering the context; and it must share new context information that it has perceived or aggregated (if any), in order to make it available to other components of the system.

The purpose of this chapter is to present a model that makes possible the isolation of application-independent context-related processes in a specific layer – a middleware that relies on a simple, flexible and generic context representation in order to perform tasks such as situation detection, decision, and sharing of context information.

Moreover, in order to enable simple situation detection and action without the need for domain-specific processes, context information and recognized situations should be easy to view, edit and manage by the user directly.

Throughout this chapter we will be using, as example, an Ambient Assisted Living (AAL) scenario: Emily is an elderly woman that lives alone. Most of her activity happens indoors, but some days she goes shopping outside of the house. Her caretakers have configured an AmI system that uses motion sensors, RFID tags and AI to detect potential emergency situations and assist Emily in her daily life.

After discussing some related work in the fields of MAS for Ambient Intelligence, and of context-awareness in AmI, we elaborate on our perspective on the problem in Section 3. Section 4 will present the model of a context-awareness layer for AmI. The practical experience with implementing this model is detailed in Section 5. The last section draws the conclusions.

2 Related Work

In the field of agent-based Ambient Intelligence platforms, there are two main directions of development: one concerning agents oriented toward assisting the user, based on centralized repositories of knowledge (ontologies) and complex platforms [14, 10], and one concerning the coordination of agents associated with devices, sometimes using agent mobility, in order to resolve complex tasks that no agent can do by itself, also considering distributed control and fault tolerance [12, 3]. In both approaches context-aware reasoning is lacking, or is performed in centralized repositories, away from the agents in need of the reasoning process. We propose a model in which reasoning can be done in the agent, close to the user, not depending on centralized components.

In context-awareness for pervasive computing, infrastructures for the processing of context information [11, 9] contain several layers, going from sensors to the application. This type of infrastructures is useful when the context information comes from the environment and refers to environmental conditions such as location, temperature, light or weather, also having a simpler representation. Our approach is directed towards an infrastructure that is decentralized, in which each entity / agent has knowledge about the context of its user [**?**].

The research group of Diane Cook working on activity detection proposes a method that bears much similarity to our own, in the sense that textual pattern detection is used to detect behavioral patterns in activity data recorded as text [5]. By comparison, this work is directed towards activity recognition (rather than detection) and proposes a representation for context / situation information that is easier to read and handle by the carer of the assisted person.

Modeling of context information uses representations that range from tuples to logical, case-based and ontological representations [11]. These are used to determine the situation that the user is in. However, these approaches are not flexible throughout the evolution of the system – the ontologies and rules are hard to modify on the go and in a dynamical manner. Moreover, a less informed user is unlikely to be able to modify an ontology or a set of rules based on FOPL, as opposed to a visual representation based on graph. The work of Sowa [13] on conceptual graphs is interesting with respect to our research, but in our work we focus on graph matching for situation detection.

3 Context-Awareness from an Application-Independent Perspective

We position this work at the intersection of Ambient Intelligence, context-awareness and multi-agent systems.

Software agents (and multi-agent systems [8]) are an appropriate paradigm for the implementation of AmI [6, 14], as agents are autonomous, therefore more reliable in a dynamic environment and more capable of focusing on just one user or one functionality. Moreover, there is a large amount of research that can be used from the fields of multi-agent platforms and agent reasoning and communication. Even if an AmI system does not use agents, reliability and robustness concerns call for distributed, autonomous application components that are much like software agents.

As most of the agents in an AmI system need to access context information that is relevant to their activity, many features of context-aware behavior can be integrated in a generic, application-independent layer below the main application functionality, to serve as component in a middleware for context management. Such a **context-**

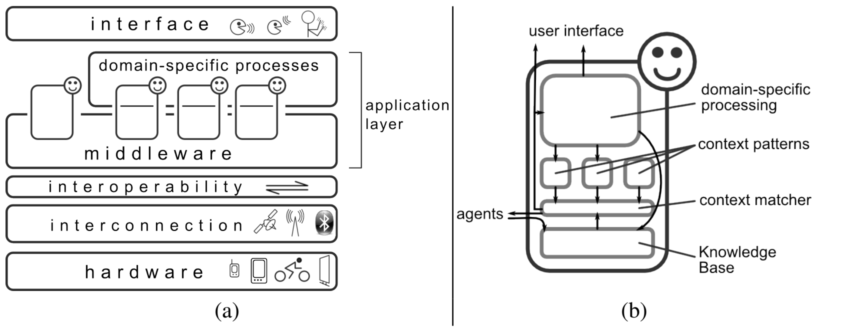


Fig. 1 (a) A layered view of an Ambient Intelligence system, dividing the application layer into context-awareness middleware and domain-specific processes. (b) A schematic view of an agent's internals, presenting the context graph (KB), matcher and patterns, and domain-specific processing.

awareness middleware would handle context information transfer, detection of information that is relevant to the application, a certain range of context-aware decisions, and sharing of new context information with other agents. This approach is directed towards a decentralized solution (supporting robustness and dependability) in which all context information is stored in the agents to which that information is potentially relevant at the current time (the context is the "dressing" of the agent's focus [2]).

An Ambient Intelligence system can be modeled as having several layers (see Figure 1 (a)) [7]: the devices that compose the system; the pervasive network connecting the devices; an interoperability layer ensuring uniform representations throughout; the application layer, concerned with intelligent behavior and application logic; and the multi-modal natural user interface. In this model, the context-awareness middleware fits inside the application layer, underlying domain-specific processes. Ideally, the middleware handles all incoming and outgoing communication between agents, providing applications the information relevant to their activity. There may be agents that don't even have any application-specific logic, and rely only on the functionality offered by the middleware to provide data to the interface.

By using the underlying middleware, the application must be able to **access context information** (e.g. know about the activity of the user), must **understand the context** (e.g. understand the relations between the different facts and to evaluate the relevance of a piece of information) and must be able to **decide upon correct context-aware action** (e.g. know about the user's experience and expectations, detect appropriate action and also be able to perform the action). Most of these features can rely on the functionality of the middleware. Its architecture is presented in the next section.

4 The Context-Awareness Layer

The middleware presented in this chapter offers to AmI applications that use it a possibility of integrating context-awareness, by providing them with storage of context information, detection of situations specified by the application, suggesting potential action, and sharing with other devices and applications context information that is potentially interesting to them.

In order to make these processes generic, an appropriate representation for context information is needed, such that it is possible to represent the user's situation and also to be able to say if the user is in a specified situation or not. For this purpose, the model that we propose uses context graphs to represent information and context patterns to represent interesting situations [**?**]. Context graphs and patterns have been developed specifically for a general, non-restrictive representation of information that allows matching patterns against graphs. Context matching can be used to compute the relevance of incoming information, to detect situation and appropriate action, and to extract information potentially relevant to other agents.

Beside the internal representation of context, the model also specifies that agents only communicate with other agents that share some context, creating a topology of the agent system that is an overlay of the actual network topology. This helps efficiency and privacy.

4.1 Formal Model

This section presents the model that the context-awareness middleware – integrated in a multi-agent system – relies on. In this model, the multi-agent system is organized on three levels (see Figure 2): containers (or machines), agents, and knowledge / context information. Each of these levels is modeled as a graph: the Container Graph shows what containers can communicate directly with each other; the Agent Graph specifies which agents share context (see Section 4.3), and what is their relation; and each agent contains a ContextGraph with the information relevant to its activity).

The **Tri-Graph** is formed by the reunion of $ContainerGraph = (Containers, Connections)$, $AgentGraph = (Agents, AgentRelations)$; and agents' context graphs CG_{Agent}:

$Tri\text{-}Graph = (Nodes, Edges)$, where

$Nodes = Containers \cup Agents \cup \bigcup_{A\ agent} CG_A.V$

$Edges = Connections \cup AgentRelations \cup AgentLocations \cup \bigcup_{A\ agent} CG_A.E$

$CG_A = (V,E)$, where $V \subseteq Concepts$ and $E = \{edge(from, to, value, persistence) \mid from, to \in Concepts, value \in Relations, persistence \in (0,1]\ \}$

In a **context graph** CG_A, the elements of $Concepts$ and $Relations$ are strings or URIs; $Relations$ also contains the empty string, for unnamed relations. The $value$

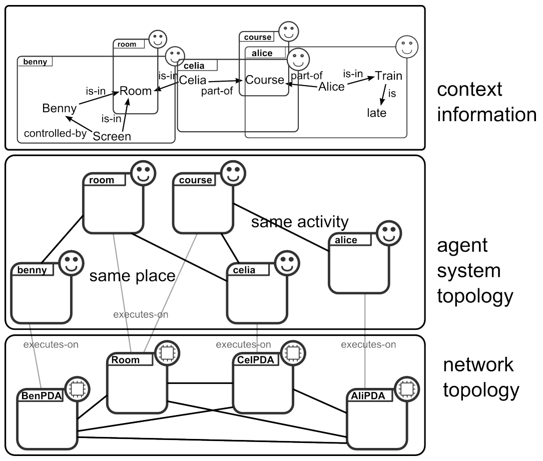


Fig. 2 A visual representation of the various graphs, in the modeling of an example scenario involving 5 agents and 4 machines.

attribute is the label of the edge. The *persistence* attribute specifies how long the edge will persist after it has been added to the context graph.

Situation recognition is done by means of **context patterns**. A pattern represents a set of associations that are specified by the user, the applications, or are extracted by the agent from the history of context information.

A pattern is also a graph, but there are several additional features that make it match a wider range of situations. For instance, some nodes may be labeled with "?" and are generic; also, edges may be labeled regular expressions (matching series of edges in the context graph.

Each agent has a set of context patterns that it matches against its context graph and against the information that it receives, in order to determine relevant situations and solve potential problems:

$Patterns = \{(G_s^P, relevance, persistence) \mid s \in PatternNames,\ G_s^P$ a graph pattern, $relevance,\ persistence \in (0,1]\}$.

A **graph pattern** is a graph $G_s^P = (V_s^P, E_s^P)$ with:

$V_s^P = \{v_i^P \mid v_i^P.label \in Concepts \cup \{?\}\}$

$E_s^P = \{(from, to, label, characteristic, actionable) \mid from,\ to \in V_s^P,\ label \in \mathscr{R}egexps(Relations),\ characteristic,\ actionable \in (0,1]\}$,

The *characteristic* feature of a pattern edge influences the measurement of how well a pattern matches a subgraph. Its *actionability* feature measures how correct it would be for the agent to infer the existence of this edge in case of a partial match between the pattern and the context graph. The *relevance* of a pattern shows how important is an information matching the pattern; *persistence* shows for how long new information will persist after being matched by the pattern. Once a pattern has been matched, its *persistence* value will be assigned to all edges in the matched subgraph. Labels in both context graphs and patterns are unique (with the exception that there may be more than one generic node in a pattern).

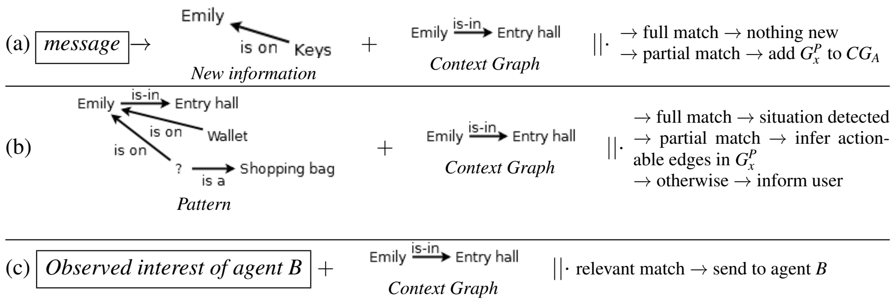

(a)
message
Emily
is on
Keys
New information
+
Emily
is-in
Entry hall
Context Graph
||·
→ full match → nothing new
→ partial match → add G_x^P to CG_A
(b)
Emily
is-in
Entry hall
is on
Wallet
is on
?
is a
Shopping bag
Pattern
+
Emily
is-in
Entry hall
Context Graph
||·
→ full match → situation detected
→ partial match → infer actionable edges in G_x^P
→ otherwise → inform user
(c)
Observed interest of agent B
+
Emily
is-in
Entry hall
Context Graph
||· relevant match → send to agent B

Fig. 3 Processes happening in a context-aware agent: (a) integrating new information; (b) detecting situation; (c) sharing information with other agents. The matching has been marked with $||\cdot$.

A **match** i between a pattern G_s^P and an agent A's context graph CG_A is defined[3] as $M_{A\text{-}si}(G'_A, G_m^P, G_x^P, f, k_f)$.

G'_A, G_m^P, G_x^P are graphs[4], with $G'_A \subseteq CG_A$, $G'_A = (V', E')$, $G_m^P = (V_m^P, E_m^P)$, $G_x^P = (V_x^P, E_x^P)$, where $V_m^P \cap V_x^P = \emptyset$, $V_m^P \cup V_x^P = V_s^P$, $E_m^P \cap E_x^P = \emptyset$ and $E_m^P \cup E_x^P = E_s^P$

That is, $G'_A \subseteq CG_A$ is a full match for the *solved part* G_m^P of pattern G_s^P. What is left of the pattern is the *unsolved part* G_x^P (also called *the problem*). There is no intersection between the solved and unsolved parts of the patterns (no common nodes or edges).

The matching function $f : V_m^P \to V'$ establishes a correspondence between the vertices of the solved part and the match in the graph with the conditions that every non-generic vertex from the solved part must match a different vertex from G'_A; every non-*RegExp* edge from the solved part must match an edge from G'_A; every *RegExp* edge from the solved part must match a chain of edges from G'_A; and G'_A does not contain other nodes or edges than the ones that are matched by the pattern (G'_A is minimal).

The number $k_f \in (0, 1]$ indicates how well the pattern G_s^P matches G'_A in match $M_{A\text{-}si}$, and is given by the normalized sum of the *characteristic* factors of matched edges, i.e.

$$k_f = \sum_{e_i^P \in E_m^P} e_i^P.characteristic \;/\; \sum_{e_j^P \in E_s^P} e_j^P.characteristic$$

Equivalently, we can define the match of any 2 graphs G_X and G_Y – where G_Y is the "pattern" – as $M_{G_X\text{-}G_Y i}(G'_X, G_m^P, G_x^P, f, k)$, since a graph is a particular case of graph pattern.

[3] There may be multiple matches between the same pattern and graph.

[4] G_x^P is not a proper graph, as it may contain edges without containing their adjacent vertices.

4.2 Reasoning

Based on the formal model presented in the previous section, there are three processes that occur constantly in the context-awareness component of the agent: the agent is able to **integrate new information** coming from other agents (e.g. an RFID reader notifies Emily's activity manager that Emily has her keys with her); the agent is able to **detect situation and act upon it** (e.g. Emily's activity manager infers that Emily is going to go out and notifies her she should take a shopping bag); and the agent is able to **share information with its neighbors** (e.g. Emily's activity manager informs her caretaker that she will go shopping). We have previously shown that these processes (behaviors) are essential and sufficient to ensure that interesting information reaches the potentially interested agents..

Whenever the agent receives a message from which a graph can be extracted, the graph is matched against the context graph of the agent. If there is no match, it means that the information has no relevance with respect to the agent's activity, and it is discarded. If there is a full match, the agent already has the information, so no change occurs. If there is a partial match, the new information is integrated with the agent's context graph, by simply adding the unsolved part of the match to the context graph. This way, the agent acquires new information (see the example in Figure 3 (a)).

Whenever the context graph of the agent changes, related patterns of the agent are matched against the context graph, to detect if the situation of the agent has changed. In case of a full match, the situation is considered as current. In case of a partial match, the actionability of the edges in the unsolved part is checked, and if it is greater than the k_f of the match, the edges are added to the context graph. This is how the agent infers new information. Otherwise, depending on the k_f, the user may be notified of the partial match, as user action may be needed and action cannot be taken by the agent autonomously (see the example in Figure 3 (b)).

As the agent receives information from other (neighbor) agents, it forms an 'observed interest' record for the agents. Since agents send only information that is interesting to them, extracting interest indication from received messages may be useful. This indication can be represented as patterns of the agent that may be interesting for the other agent. Whenever information matching those patterns is found, it will be sent to the other agent, as it may be potentially interesting to it (see the example in Figure 3 (c)).

An important process in the agent is the removal of parts from the context graph – **forgetting outdated information**. For example, if the system detected that Emily was in the kitchen 5 minutes ago and her position has not been reconfirmed since, it may well be that she has moved elsewhere and that information can be considered as obsolete, as a new detection should take its place. When new edges are added, their *persistence* is set according to the indications of the pattern that contains the edge. With time, persistence of the edges in the context graph of the agent (CG_A) fades, and as it reaches zero, the edge is removed (along with any resulting isolated nodes).

Table 1 The possible relations between different agent types, resulting from the mapping of context to system topology.

Agent type (context type)	Possible incoming relations (and their sources)	Possible outgoing relations (and their destinations)
Place (spatial)	*is-in* (← *Activity*, *User*, *Device*, *Service*, *Place*)	-
Activity (activity)	*part-of* (← *User*, *Group*, *Activity*, *Service*)	*of* (→ *User*)
Device (computing)	*executes-on* (← *Service*)	*is-in* (→ *Place*), *controlled-by* (→ *User*)
Service (computing)	-	*executes-on* (→ *Device*), *is-in* (→ *Place*), *part-of* (→ *Activity*)
User (social)	*controlled-by* (← *Device*), *of* (← *Activity*), *connected-to* (← *User*)	*part-of* (→ *Activity*), *in* (→ *Group*), *connected-to* (→ *User*)
Group (social)	*in* (← *User*)	*part-of* (→ *Activity*)
Organization (social)	*part-of* (← *User*, *Group*, *Place*, *Activity*)	*part-of* (→ *Organization*)

All actions that an agent can take are related to matched patterns and added edges (creation of relations between concepts). The actual actions that are connected to the addition of edges are performed by attached procedures that are part of the application-specific part of the agent, allowing the agent to actually change its environment accordingly. This is why the only edges that are ”actionable” should be the ones that correspond to effects that the agent can actually create.

4.3 Dynamic Agent Topology

Our approach is directed towards the context-aware transfer of information in a distributed, decentralized MAS for Ambient Intelligence. But to achieve communication efficiency, and to deal with privacy concerns, we have devised for our model an agent topology that is induced by context. In this **context-based topology**, if two agents share context, then they should be neighbors. The topology becomes an overlay on the actual network that the agents use to communicate. Shared context can be a common activity, a common place, etc.

There are five types of context that we consider for the agent hierarchy, four being identified by Chen and Kotz [4], and the fifth being activity, an important aspect in the association-based model of Henricksen et al [9]. The five types are **spatial context, temporal context, computational context, activity context and social context**. For each aspect of context we introduce agent types and relations between agents. The possible relations between agents are presented in Table 1. While temporal context is an aspect of context that we consider, we do not have a specialized agent for time intervals (which would be the hierarchical element of temporal context), as an agent that manages a time interval does not make much sense: since the internal context representation, as well as the relations between agents, reflect the present situation – therefore shared temporal context is already achieved. The temporal aspect is further discussed in Section 4.4.

Using such a topology has several advantages: the agents only send information to agents that share context with them – an agent would not find any interest in information received from an agent with absolutely no common context with it; and

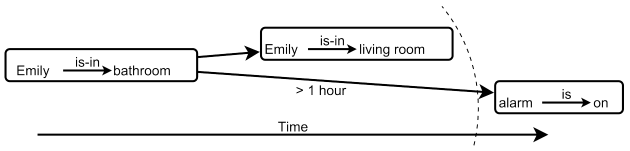


Fig. 4 Example of *Timeline* of an agent. The dotted line represents the current moment of time.

when looking for information, the search will be kept local (in terms of context), there where it makes more sense (and is more likely to yield results).

Moreover, in previous work we have explored the idea of mapping context structure to agent hierarchies [**?**]. Most aspects of context are hierarchical: places are parts of other places, activities are part of more general activities, computational resources belong to places or are related to certain activities, social structure is hierarchical, etc. Using hierarchies not only helps us organize the system, but allows us to use hierarchical mobility – in which a mobile agent moves together with its subtree of agents.

Whenever the context of an agent changes, the relations with the other agents change as well – the topology of the system is dynamic. In some cases, mobile agents may be used. For example, the *Shopping List* agent may normally execute on the same machine as the kitchen agent, but when Emily goes shopping, it should move, as a part of her *Shopping* activity, to Emily's personal device, which will stay with her. It makes sense that an agent managing an activity should reside closer to the place where the activity takes place.

The choice of the presented types of context and relations is not random, as they cover the types identified in the literature. The use of specific types of agents and relations does not reduce the generality of the model, as these types may be used for any Ambient Intelligence application. The context-aware topology is not claiming to be a complete implicit representation of context – it only helps organize the system and keep information transfer between the agents local in terms of shared context.

4.4 The Temporal Aspect

Using context graphs and patterns is very useful when working on the *current* situation. If the Context Graph represents context information about the present, then patterns can identify the situation(s) that the user is in *right now*. However, some situations depend on the passage of time. For example, if the system knows Emily is in the bathroom, it cannot tell if any problem has occurred. If Emily has been in the bathroom for 5 minutes, it is alright; if she has been there for one hour, then it is likely that there is a problem. Time-related situations such as this may be handled by introducing time moments or time intervals as nodes in the context graph, and can be handled by specialized processes in the application specific part of the agent.

However, as time-related situations are frequent in all types of AmI scenarios, we have developed a generic solution. This is our most recent research.

Beside the "instantaneous" patterns already presented in Section 4.1, an agent can also contain *Timelines*. A timeline is a second-order graph pattern: $T_s^P = (V^P, E^P)$, with $V^P \subseteq Patterns$ and $E^P = \{\ (from,\ to,\ value,\ characteristic,\ actionable)\ \mid from, to \in V_s^P,\ characteristic,\ actionable \in (0,1]\}$. The label of an edge can take values that are time lengths (e.g. "5 minutes", "more than 1 hour", etc), special values (e.g. *next*) or the empty string. Timelines are restricted to be single root directed acyclic graphs, so that they can represent branching paths of temporal events. Once a timeline is activated, its nodes and edges describe a sequence in which patterns should be matched.

A particular timeline becomes active when the pattern in the root of the timeline is matched. The edges going out from the root become *enabled*. Enabled edges can become active depending on their value. When the pattern at the destination of an active edge is matched, all other edges are inactivated and disabled and the edges outgoing from the matched node are enabled. The process continues until no edges are enabled or active, in which case the timeline is inactivated. Edges with no value are activated immediately and remain so indefinitely (until they are disabled by external events); edges labeled with "less than *time*" become active immediately and are inactivated after $time$; edges labeled with *next* become inactive when their source node is not matched anymore and their destination node is not matched immediately after; edges specifying precise times, or lower limits on times ("more than 30 minutes") become active only after the specified amount of time passes.

In the example in Figure 4, the simple timeline specifies that if Emily entered the bathroom and has not come back to the living room (which is next to the bathroom) in under 1 hour, then the alarm must be activated. Using timelines is a simple and easy to visualize method to specify possible sequences of events.

5 Practical Experience

The model that we have presented in the previous section has been developed together with the experience of several software projects using agents for integrating context-awareness in Ambient Intelligence applications. This section will highlight some of the practical results in this experience.

5.1 Agent Behavior and Topology

The described agent behavior and topology have been designed through experiments using platforms described in previous work [**?**].

Agent behavior has been studied through a large number of simulations using the AmIciTy:Mi platform, that allows for fast simulations of large numbers of agents, using scenario files that completely specify the evolution of the simulation[5]. Simulations of up to over 1000 agents have been used to study the context-aware dissemination of information based on local communication, relying on self-organization mechanisms. The leading principle of the best behavior was that agents should send information that is interesting to them to neighbor agents that are potentially interested in that information.

The context-based hierarchical **agent topology** described in Section 4.3 improves the one in AmIciTy:Mi experiments and has been validated through experiments using the agent-oriented programming (AOP) language CLAIM, based on ambient calculus, that supports hierarchical mobile agents. A distributed AmI scenario has been publicly demonstrated [**?**].

5.2 Context Representation and Matching

As all of the processes in the context-awareness middleware that we present rely on matching graphs, we have developed a purpose-build, efficient graph matching algorithm, dedicated especially to directed graphs in which most edges are labeled[6] [**?**].

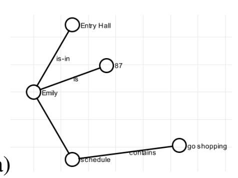
The **context-matching algorithm** is focused on matching edges. It works with valid partial matches of various sizes and merges them in order to form larger (better) matches. The algorithm has two phases. First, it generates a set of all possible single-edge matches. Then, it selects pairs of compatible matches that it merges in order to create new matches. The search for new matches is close to a depth-first search, in order to get better matches faster. The gist of the algorithm is that it does not test the compatibility (from the point of view of merging) of the matches before each merger, but instead uses for each match a set of data structures (a frontier, a set of adjacent, “immediate”, merger candidates and a set of non-adjacent, “outer” merger candidates) that allow the algorithm to know precisely if two matches can be merged or not.

Single-edge matches are compatible if their pattern edges have any common vertices and if those vertices correspond to the same node in the matched graph. The match could be potentially merged, later, with any other match that is not adjacent to it. In the second phase of the algorithm, for each match, new matches are created by merging it with immediate merger candidates (guaranteed to be compatible). When matches M_1 and M_2 are merged, the newly created M match has candidate sets that are guaranteed to be correct without actually checking them. The set of immediate merger candidates of M is the union of immediate merger candidates for both M_1

[5] We thank Cristian Gratie, Guillaume Masson, Alexandre Hocquard and Sofia Neatză for their contributions to the project.
Code is open source at `https://github.com/andreiolaru-ro/AmIciTy-Mi`.

[6] Code is open source at `https://github.com/andreiolaru-ro/net.xqhs.Graphs`.

(a)



(b) $\mathit{Emily}(\rightarrow \mathit{schedule} \xrightarrow{\mathit{contains}} \mathit{go\ shopping})(\xrightarrow{\mathit{is}} 87) \xrightarrow{\mathit{is\text{-}in}} \mathit{Entry\ Hall}$

(c)
```
Emily (-> schedule-contains -> go shopping)
      (-is-> 87) -is-in-> Entry Hall
```

Fig. 5 Various representation of a graph: (a) graphical; (b) textual; (c) ASCII.

and M_2, immediate candidates for M_1 that are outer candidates for M_2 and immediate candidates for M_2 that are outer candidates for M_1. The outer merger candidates are candidates that were acceptable by both M_1 and M_2 but were not immediate merger candidates for any of the two.

Both the experimental results and the complexity analysis [**?**] have shown that the algorithm is particularly appropriate for the problem of context matching, yielding good results for graphs that come from AmI scenarios. As an additional feature, the algorithm can be stopped at any time, as with time it builds better and better matches. Not waiting for the algorithm to complete does not mean that no results are obtained, only that they may not be the best.

Representation Features have been developed so that context graphs and patterns in the agent can be visualizable and editable by the user directly, even without application-specific processes. First, we have developed a linear textual representation of directed graphs, for the purpose of displaying a human-readable form of graphs in the output console and to easily input graphs from the keyboard. It uses vertex and edge names, arrows, stars and parentheses to completely display a graph. Each edge is shown once, and nodes are repeated once per graph cycle. For instance, a graph that is formed of three nodes (A, B, C) linked by two edges a and b is represented as $A \xrightarrow{a} B \xrightarrow{b} C$; the cycle ABC is represented as $A \rightarrow B \rightarrow C \rightarrow *A$; a tree with root A and children B and C is represented as $A(\rightarrow B) \rightarrow C$. This representation is also easy to copy and paste between various applications on desktop PCs or mobile devices. Based on the linear textual representation, we have also built a graphical representation for graphs and for matches. An example is shown in Figure 5.

A **Continuous Context Matching Platform** has been developed, that allows for one context graph to be matched against a large set of graph patterns, but avoiding that a full graph matching process is carried out at every change in the graph. Instead, intermediate matches are kept in memory so that at any modification to the context graph the algorithm creates only new matches that are based on an added edge, or removes matches that include a deleted edge. This platform can be used by an agent to assist it in its context-aware behavior (as presented in Section 4.2.

5.3 Context-Based Agent Platform

In order to further test the model that we have developed for a context-awareness middleware for AmI applications, we have integrated context-awareness features in a platform for building and running AmI applications – the tATAmI platform (towards Agent Technologies for Ambient Intelligence)[7] [**?**]. The platform was implemented using a modular structure, and features tools for the visualization and tracking of agents, as well as for the realization of repeatable experiments, based on scenario files. The platform is underpinned by JADE[8] for communication, management and mobility features.

The platform allows the implementation of various AmI applications and is meant to validate the model presented in Section 4 through the integration of all of its components. The platform uses an evolved version of the CLAIM language, called S-CLAIM, which is simpler and easier to use. The definition of agents is based on behaviors, which can be reactive or proactive. The agents use Context Graph Knowledge Bases for context information that are accessed by means of a small number of functions that use patterns represented in text to locate information in the Knowledge Base. Agents also feature hierarchies and hierarchical mobility is implemented, allowing for the definition of the context-aware topology.

6 Conclusions and Future Work

This chapter presents a model in which generic functionality related to context-awareness in AmI applications can be isolated to a layer below domain-specific processing. In an agent-based architecture, the context-aware middleware lives in the 'lower' part of the agent, handling incoming context information, situation detection and context information sharing.

The model relies on a representation for context that is based on graphs, and patterns that are matched against the current context to detect interesting information and current situation. Outside the agent, an agent topology can be defined that reflects the actual context of the agent. The model of context patterns can be extended to handle temporal relations and sequences of events.

Practical experience with the model includes an algorithm for persistent context matching involving multiple patterns, graphical and textual representations for graphs and patterns that are easy to read and to input, as well as a platform for context-aware AmI applications.

Future work involves the implementation and simulation of more complex scenarios, as well as the deployment of the tATAmI platform in the Ambient Intelli-

[7] We thank Thi Thuy Nga Nguyen, Marius-Tudor Benea, Emma Sevastian, prof. Amal El Fallah Seghrouchni, and Cédric Herpson for their contributions to the project. The code is open source and can be found at `https://github.com/tATAmI-Project`.

[8] Java Agent Development Framework `http://jade.tilab.com/`.

gence Laboratory of our Faculty, to serve as a context-aware middleware for AmI applications. Temporal elements in context matching are still at the beginning, and more work is required to cover a wider range of time-related issues and scenarios.

Acknowledgements This work has been supported by the Sectoral Operational Programme Human Resources Development (POS DRU) 2014-2020 of the Romanian Ministry of Labor, Family, Social Protection and Elderly.

References

1. Augusto, J.C., McCullagh, P.J.: Ambient intelligence: Concepts and applications. Computer Science and Information Systems (ComSIS) **4**(1), 1–27 (2007)
2. Brézillon, J., Brézillon, P.: Context modeling: Context as a dressing of a focus. In: B. Kokinov, D. Richardson, T. Roth-Berghofer, L. Vieu (eds.) Modeling and Using Context, *Lecture Notes in Computer Science*, vol. 4635, pp. 136–149. Springer Berlin Heidelberg (2007). DOI 10.1007/978-3-540-74255-5_11. URL `http://dx.doi.org/10.1007/978-3-540-74255-5_11`
3. Cabri, G., Ferrari, L., Leonardi, L., Zambonelli, F.: The LAICA project: Supporting ambient intelligence via agents and ad-hoc middleware. Proceedings of WETICE 2005, 14th IEEE International Workshops on Enabling Technologies, 13-15 June 2005, Linköping, Sweden pp. 39–46 (2005)
4. Chen, G., Kotz, D.: A survey of context-aware mobile computing research. Technical Report TR2000-381, Dartmouth College (2000)
5. Cook, D.J., Krishnan, N.C., Rashidi, P.: Activity discovery and activity recognition: A new partnership. Cybernetics, IEEE Transactions on **43**(3), 820–828 (2013)
6. Ducatel, K., Bogdanowicz, M., Scapolo, F., Leijten, J., Burgelman, J.: Scenarios for ambient intelligence in 2010. Tech. rep., Office for Official Publications of the European Communities (2001)
7. El Fallah Seghrouchni, A.: Intelligence ambiante, les defis scientifiques. presentation, Colloque Intelligence Ambiante, Forum Atena (2008)
8. Ferber, J.: Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
9. Henricksen, K., Indulska, J.: Developing context-aware pervasive computing applications: Models and approach. Pervasive and Mobile Computing **2**(1), 37–64 (2006)
10. Lech, T.C., Wienhofen, L.W.M.: AmbieAgents: a scalable infrastructure for mobile and context-aware information services. Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), July 25-29, 2005, Utrecht, The Netherlands pp. 625–631 (2005)
11. Perera, C., Zaslavsky, A., Christen, P., Georgakopoulos, D.: Context aware computing for the internet of things: A survey. IEEE Communications Surveys and Tutorials **16**(1), 414–454 (2013)
12. Rogers, A., Corkill, D., Jennings, N.: Agent technologies for sensor networks. IEEE Intelligent Systems **24**(2), 13–17 (2009)
13. Sowa, J.: Conceptual graphs. Foundations of Artificial Intelligence **3**, 213–237 (2008)
14. Tapia, D., Abraham, A., Corchado, J., Alonso, R.: Agents and ambient intelligence: case studies. Journal of Ambient Intelligence and Humanized Computing **1**(2), 85–93 (2010)