A Rule-based Contextual Reasoning Platform for Ambient Intelligence environments

Assaad Moawad[1], Antonis Bikakis[2], Patrice Caire[1], Grégory Nain[1] and Yves Le Traon[1]

[1] University of Luxembourg, SnT
firstname.lastname@uni.lu
[2] Department of Information Studies, University College London
a.bikakis@ucl.ac.uk

Abstract. The special characteristics and requirements of intelligent environments impose several challenges to the reasoning processes of Ambient Intelligence systems. Such systems must enable heterogeneous entities operating in open and dynamic environments to collectively reason with imperfect context information. Previously we introduced Contextual Defeasible Logic (CDL) as a contextual reasoning model that addresses most of these challenges using the concepts of *context*, *mappings* and *contextual preferences*. In this paper, we present a platform integrating CDL with Kevoree, a component-based software framework for Dynamically Adaptive Systems. We explain how the capabilities of Kevoree are exploited to overcome several technical issues, such as communication, information exchange and detection, and explain how the reasoning methods may be further extended. We illustrate our approach with a running example from Ambient Assisted Living.

Keywords: contextual reasoning, distributed reasoning, Ambient Intelligence, system development

1 Introduction

Ambient Intelligence (AmI) constitutes a new paradigm of interaction among agents acting on behalf of humans, smart objects and devices. Its goal is to transform our living and working environments into *intelligent spaces* able to adapt to changes in contexts and to their users' needs and desires. This requires augmenting the environments with sensing, computing, communicating and reasoning capabilities. AmI systems are expected to support humans in their every day tasks and activities in a personalized, adaptive, seamless and unobtrusive fashion [1]. Therefore, they must be able to reason about their *contexts*, i.e. with any information relevant to the interactions between the users and system.

The imperfect nature of context, and the special characteristics of AmI environments impose several challenges in the reasoning tasks. Henricksen and Indulska [2] characterize four types of imperfect context: *unknown*, *ambiguous*, *imprecise*, and *erroneous*. Sensor or connectivity failures, which are inevitable

in wireless connections, result in situations that not all context data is available at any time. When data about a context property comes from multiple sources, then context may become ambiguous. Imprecision is common in sensor-derived information, while erroneous context arises as a result of human or hardware errors. Context is typically distributed among agents with different views of the environment that use different languages to describe it. Due to the highly dynamic and open nature of the environments and the unreliable wireless communications, agents do not typically know a priori all other entities present at a specific time instance, nor can they communicate directly with all of them. Despite these restrictions, they must be able to reach common conclusions about the state of the environment and collectively take context-aware decisions.

In previous works we introduced a new nonmonotonic logic, *Contextual Defeasible Logic* (CDL), tailored to the specific requirements of AmI systems. We presented its language and argumentation semantics, proved its formal properties [3], and developed algorithms for distributed query evaluation [4]. We also showed how CDL may enable heterogenous devices to collectively reason with imperfect context, and take context-aware decisions in a distributed fashion. This paper is a further step towards reasoning with CDL and making it real in an AmI environment. Here, we focus on scenarios from *Ambient Assisted Living* (AAL), though the same findings may be applied to any subfield of AmI. AAL is a relatively new research and application domain focused on the technologies and services to enhance the quality of life of people with reduced autonomy, such as the elderly. In the framework of the CoPAInS project[1], such problematic is studied to evaluate the tradeoffs to be made in AAL systems [5], particularly as they pertain to conviviality, privacy and security [6]. Creating this bridge between CDL and AAL requires addressing issues, such as dynamicity, adaptability, detection and communication between devices, and is therefore not trivial. This prompts our research question: *How to deploy CDL in real AmI environments?*

To address this question, we created a platform that maps the CDL model to the business view of AAL using Kevoree [7]. Kevoree is designed to facilitate the development of Distributed Dynamically Adaptive Systems. Particular features of Kevoree make it a suitable for our needs: ability to implement heterogeneous entities as independent *nodes*; use of communication *channels* to enable message exchanges between nodes; support for shared models to enable common representations for different types of nodes; adaptive capabilities to fit with the open and dynamic nature of AAL.

The contribution of this work is twofold: (a) we describe solutions for the deployment of a theoretical model (CDL) in real AmI environments; and (b) we provide an AAL platform, usable by anyone to test and implement AAL scenarios. The rest of the paper is structured as follows. Section 2 describes our running AAL example. Sections 3 and 4 present the theoretical and technical background of this work, namely CDL and Kevoree. Section 5 describes the integration of CDL with Kevoree in our novel AAL platform. Section 6 presents related work, and Section 7 concludes and presents our plans for future work.

[1] http://wwwen.uni.lu/snt/research/serval/projects/copains

2 An Ambient Assisted Living Example

In this section, we present an Ambient Assisted Living (AAL) scenario, part of a series of scenarios validated by HotCity, the largest WI-FI network in Luxembourg, in the Framework of our CoPAInS project (Conviviality and Privacy in Ambient Intelligence Systems).

In our scenario, visualized in Figure 1, the eighty-five years old Annette is prone to heart failures. The hospital installed a Home Care System (HCS) at her place. One day, she falls in her kitchen and cannot get up. The health bracelet she wears gets damaged and sends erroneous data, e.g., heart beat and skin temperature, to the HCS. Simultaneously, the system analyzes Annette's activity captured by the Activity Recognition Module (ARM). Combining all the information to Annette's medical profile, and despite the normal values transmitted by Annette's health bracelet, the system infers an emergency situation. It contacts the nearby neighbors asking them to come and help.

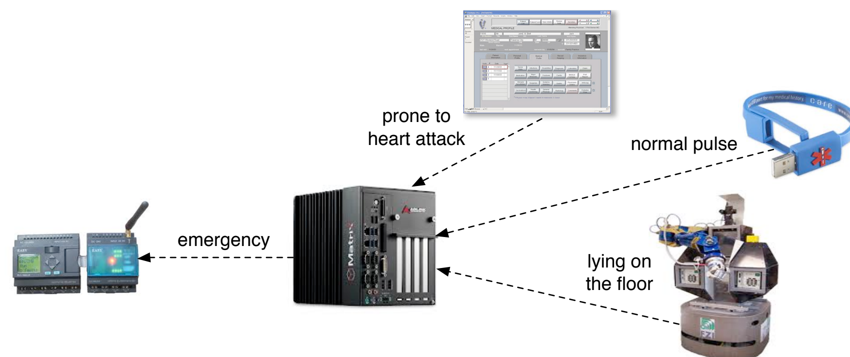

Fig. 1. Context Information flow in the scenario.

This scenario exemplifies challenges raised when reasoning with the available context information in Ambient Intelligence environments. Furthermore, it highlights the difficulties in making correct context-dependent decisions.

First, context knowledge may be erroneous. In our example, the values transmitted by the health bracelet for Annette's heart beat and skin temperature, are not valid, thereby leading to a conflict about Annette's current condition. Second, local knowledge is incomplete, in the sense that none of the agents involved has immediate access to all the available context information. Third, context knowledge may be ambiguous; in our scenario, the HCS receives mutually inconsistent information from the ARM and the health bracelet. Fourth, context knowledge may be inaccurate; for example, Annette's medical profile may contain corrupted information. Finally, devices communicate over a wireless network. Such communications are unreliable due to the nature of wireless networks, and are also restricted by the range of the network. For example, the health bracelet may not be able to transmit its readings to HCS due to a damaged transmitter.

In the next section, we analyze how CDL enables devices to model and reason with such imperfections.

3 Contextual Defeasible Logic

Contextual Defeasible Logic (*CDL*) is a distributed rule-based approach for contextual reasoning, which was recently proposed as a reasoning model for Ambient Intelligence systems [3]. CDL adopts ideas from:

- *Defeasible Logic* [8] - it is rule-based, skeptical, and uses priorities to resolve conflicts among rules;
- *Multi-Context Systems* (MCS, [9, 10]) - logical formalizations of distributed context theories connected through mapping rules, which enable information flow between contexts. In MCS, a *context* can be thought of as a logical theory - a set of axioms and inference rules - that models local knowledge.

Below, we present the representation model of CDL and explain how it fits with the special characteristics and requirements of Ambient Intelligence environments. We also present an operational model of CDL in the form of a query evaluation algorithm, which we implemented in Kevoree.

3.1 Representation Model

In CDL, the MCS model is extended with defeasible rules and a preference relation on the system contexts. In CDL, a MCS C is a set of contexts C_i: A context C_i is defined as a tuple of the form (V_i, R_i, T_i), where V_i is the vocabulary of C_i, R_i is a set of rules, and T_i is a preference ordering on C.

V_i is a set of positive and negative literals of the form $(c_i : a_i)$ and $\sim (c_i : a_i)$, which denotes the negation of $(c_i : a_i)$. Each context uses a distinct vocabulary, i.e. $V_i \cap V_j = \emptyset$ *iff* $i \neq j$. This reflects the fact that each entity (e.g. device in an Ambient Intelligence environment) may use its own terminology. We should note, though, that the proposed model may also enable different contexts to use common literals (e.g. URIs) by adding a context identifier, e.g. as a prefix, in each such literal and using appropriate mappings to associate them to each context.

R_i consists of a set of *local rules* and a set of *mapping rules*. The body of a local rule is a conjunction of *local* literals (literals that are contained in V_i), while its head is labeled by a local literal. There are two types of local rules:

- Strict rules, of the form

  $$r_i^l : (c_i : a^1), \ldots, (c_i : a^{n-1}) \rightarrow (c_i : a^n)$$

  They express sound local knowledge and are interpreted in the classical sense: whenever the literals in the body of the rule $((c_i : a^1), \ldots, (c_i : a^{n-1}))$ are strict consequences of the local theory, then so is the conclusion of the rule $((c_i : a^n))$. Strict rules with empty body denote factual knowledge.
- Defeasible rules, of the form

  $$r_i^d : (c_i : a^1), \ldots, (c_i : a^{n-1}) \Rightarrow (c_i : a^n)$$

  They are used to express uncertainty: a defeasible rule cannot be applied to support its conclusion if there is adequate contrary evidence.

Mapping rules associate local literals of C_i with literals from the vocabularies of other contexts (*foreign literals*). The body of each such rule is a conjunction of local and foreign literals, while its head is labeled by a local literal. Mapping rules are modeled as defeasible rules of the form:

$$r_i^m : (c_j : a^1), \ldots, (c_k : a^{n-1}) \Rightarrow (c_i : a^n)$$

A mapping rule associates literals from different contexts (e.g. $(c_j : a^1)$ from C_j and $(c_k : a^{n-1})$ from C_k), with a local literal of the context that has defined r_i, which labels the head of the rule (here $(c_i : a^n)$). By representing mappings as defeasible rules, we can deal with ambiguities and inconsistencies caused by importing mutually conflicting information from different contexts.

Finally, each context C_i defines a strict total preference ordering T_i on C to express its confidence in the knowledge it imports from other contexts:

$$T_i = [C_k, C_l, ..., C_n]$$

C_k is preferred to C_l by C_i, if C_k precedes C_l in T_i. The strict total ordering enables resolving all potential conflicts that may arise from the interaction of contexts through their mapping rules. In a later version of CDL [11], T_i is defined as a partial preference order on C, which enables handling incomplete preference information. For sake of simplicity, we adopt here the original definition of T_i.

Example. The scenario described in Section 2 may be modeled as follows in CDL. We consider 5 different contexts: *sms* for the SMS system, *hcs* for the Home Care Syste, *arm* for the activity recognition module, *br* for the bracelet, and *med* for the medical profile. *sms* has only one mapping rule according to which, when the Home Care System detects an emergency situation, the SMS system dispatches messages to a prescribed list of mobile phone numbers:

$r_{sms}^m : (hcs : emergency) \Rightarrow (sms : dispatchSMS)$

The Home Care system imports information from the activity recognition module, the bracelet and Annete's medical profile to detect emergency situations using two mapping rules:

$r_{hcs}^{m1} : (br : normalPulse) \Rightarrow \neg(hcs : emergency)$
$r_{hcs}^{m2} : (arm : lyingOnFloor), (med : proneToHA) \Rightarrow (hcs : emergency)$

The factual knowledge of the other three modules is modeled using local rules with empty body:

$r_{br}^l :\rightarrow (br : normalPulse)$
$r_{arm}^l :\rightarrow (arm : lyingOnFloor)$
$r_{med}^l :\rightarrow (med : proneToHA)$

The Home Care System is configured to give highest priority to information imported by the medical profile and lowest priority to the bracelet:
$T_{hcs} = [med, arm, br]$

3.2 Distributed Query evaluation

In [3] we presented an argumentation semantics of CDL, while in [4] we provided four algorithms for query evaluation. *P2P_DR* is one of these algorithms, which is called when a context C_i is queried about the truth value of one of its local literals $(c_i : a_i)$, and roughly proceeds as follows:

Algorithm 1 *P2P_DR*

---

if $(c_i : a_i)$ (**or** $\neg(c_i : a_i)$) is derived as a conclusion of the local rules of C_i **then**
  return *true* (or *false* resp.)
else
  for all rules r_i in c_i that have $(c_i : a_i)$ or $\neg(c_i : a_i)$ in their heads **do**
    if r_i is applicable **then**
      Compute the Supportive Set of r_i, SS_{r_i}
  Compute the Supportive Sets of $(c_i : a_i)$, $SS_{(c_i:a_i)}$, and $\neg(c_i : a_i)$, $SS_{\neg(c_i,a_i)}$
  if $SS_{(c_i:a_i)}$ is *stronger* than $SS_{\neg(c_i,a_i)}$ with respect to T_i **then**
    return *true*
  else
    return *false*

We should note that a rule r_i is applicable when for all its body literals we have obtained positive truth values. SS_{r_i} is the union of the foreign literals with the Supportive Sets of the local literals contained in the body of r_i, while $SS_{(c_i:a_i)}$ is the *strongest* between the Supportive Sets of the rules with head $(c_i : a_i)$. A set of literals S_1 is *stronger* than set S_2 w.r.t. T_i *iff* there is a literal l in S_2, such that all literals in S_1 are stronger than l w.r.t. T_i. A literal $(c_k : a)$ is stronger than literal $(c_l : b)$ w.r.t. T_i *iff* C_k precedes C_l in T_i.

Example (continued). In our running example, a query to *sms* about $(sms : dispatchSMS)$ will initiate a second query to *hcs* about $(hcs : emergency)$. The second query will in turn initiate three more queries: a query to *br* about $(br : normalPulse)$; a query to *arm* about $(arm : lyingOnFloor)$; and a query to *med* about $(med : proneToHA)$. *P2P_DR* will return *true* for the latter three queries, and will compute $SS_{(hcs:emergency)} = \{(arm : lyingOnFloor), (med : proneToHA)\}$ and $SS_{\neg(hcs:emergency)} = \{(br : normalPulse)\}$. W.r.t. T_{hcs}, all elements of $SS_{(hcs:emergency)}$ are stronger than $(br : normalPulse)$, therefore *P2P_DR* will return a positive truth value for $(hcs : emergency)$, and the same value for $(sms : dispatchSMS)$ too, as there is no rule that supports its negation.

As shown in the example above, CDL may deal with several of the challenges of Ambient Intelligence environments, such as uncertainty, ambiguity, and erroneous data. There are still, though, some questions that need to be addressed in order to fully deploy CDL in real environments: how do the devices actually detect and communicate with each other? and how can we achieve dynamicity and adaptability? In the next sections, we describe how we addressed such questions by integrating CDL in the software platform of Kevoree.

4 Kevoree - A component based software platform

On the one hand, in Ambient Assisted Living (AAL), systems need to be adapted to users preferences and contexts. They also need to combine various data and reason about it, but the imperfect nature of context makes this task very challenging. Returning to our use case, the HCS receives data from different devices, and many situations may occur causing the data to be erroneous, e.g., Annette may have left her health bracelet next to her bed instead of wearing it, or the battery capability may be weak and preventing the bracelet from transmitting any data.

On the other hand, CDL allows to manage uncertainty and reason about it. The problem remains to apply such theoretical tools to the AAL domain in order to solve the very concrete challenges affecting patients. In this section, we present the Kevoree environment, which we use to address such issues by implementing the CDL reasoning model. This is illustrated in Figure 2.

Fig. 2. Kevoree bridges the AAL needs to the theoretical model of CDL.

4.1 Kevoree: Modeling Framework and Components

Kevoree [7] is an open-source environment that provides means to facilitate the design and deployment of Distributed Dynamically Adaptive Systems, taking advantage of Models@Runtime [12] mechanisms throughout the development process.

This development platform is made of several tools, among which the Kevoree Modeling Framework (KMF) [13], a model editor (to assemble components to create an application), and several runtime environments, from Cloud to JavaSE or Android platforms. The component model of Kevoree defines several concepts. The rest of this section describes the most interesting ones in relation to the content of this paper.

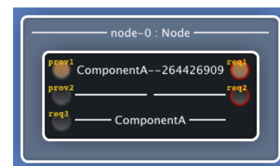

Fig. 3. A component instance, inside the node instance on which it executes

The **Node** (in grey in figure 3) is a topological representation of a Kevoree runtime. There exist different types of nodes (e.g.: JavaSE, Android, etc.) and a system can be composed of one, or several distributed heterogeneous instances of execution nodes.

Component instances are deployed and run on a node instance, as presented on figure 3. Components may also be of different types, and one or more, heterogeneous or not, component instances may run on

a single node. Components declare **Ports** (rounds on left and right sides of the component instance) for provided and required services, and input and output messages. The ports are used to communicate with other components of the system.

Groups (top shape in figure 4) are used to share models (at runtime) between execution platforms (i.e. nodes). There are different types of Groups, each of which implements a different synchronization / conciliation / distribution algorithm. Indeed, as the model shared is the same for all the nodes, there may be some concurrent changes on the same model, that have to be dealt with.

Fig. 4. An instance of Group on top, of Channel on the bottom

Finally (for the scope of this paper), **Channels** (bottom shape in figure 4) handle the semantics of a communication link between two or more components. In other words, each type of channel implements a different means to transport a message or a method call from component A to component B, including local queued message list, TCP/IP sockets connections, IMAP/SMTP mail communications, and various other types of communication.

4.2 Kevoree Critical Features

Kevoree appears to be an appropriate choice to provide solutions for the development of Ambient Intelligence systems, as it can deal with their dynamic nature. In such systems, agents are often autonomous, reactive and proactive in order to collaborate and fulfil tasks on behalf of their users.

In Kevoree, an agent is represented as a node that hosts one or more component instances. The node is responsible for the communication with other nodes by making use of the synchronization Group. Some group types implement algorithms with auto-discovery capabilities, making nodes and their components dynamically appear in the architecture model of the overall system. The fact that a new node appears in the model means that an agent is reachable, but it does not necessarily mean that it participates in any interaction. The component instances of a node provide the services for the agent. Therefore, for an agent to take part in a collaborative work, the ports of the component instances it hosts have to be connected to some ports of other agents' components.

Some features of Kevoree make it particularly suitable for our needs. First, it enables the implementation, deployment and management of heterogeneous entities as independent *nodes*. Second, it uses communication *channels* to enable the exchange of messages among the distributed components. Third, it offers a common and shared representation model for different types of nodes. Finally, it is endowed with adaptive capabilities and auto-discovery, which fit with the open and dynamic nature of AmI environments.

In the next section, we detail how we exploit the features of Kevoree and integrate them with CDL to create our AAL platform.

5 The AAL Platform

In this section, we explain how CDL and Kevoree are integrated in our AAL platform. We should note that the parts of CDL, which were not directly mapped to existing elements Kevoree, were implemented in Java.

5.1 Query Component

In our platform, the notion of context, as this is described in Section 3, is implemented by a new component type that we developed, called *Query Component*. This component has two inputs: *Console In* and *Query In*, and two outputs: *Console out* and *Query Out*. The Query Component has three properties: a *Name*, an *initial preference address* and an *initial knowledge base address*. In Kevoree, each instance must have a unique name. In our platform, we use this unique name to specify the sender or the recipient of a query. The preference address and the knowledge base address contain the addresses of the files to be loaded when the component starts. The knowledge base file contains the rule set of a context, while the preference file contains the preference order of the context implemented as a list.

Each component has two console (in/out) and two query (in/out) ports. The console input port is used to send commands to the component, e.g. to update its knowledge base or change its preference order. The outputs of the commands are sent out to the console output port. The query in/out ports are used when a component is sending/receiving queries to/from other components. Queries are sent via the "Query out" port and responses are received via "Query In".

Internally, the Query Component has some private variables, which represent its knowledge base, the preference order and a list of query servant threads currently running on it. When the component receives a new query, it creates a new query servant thread dedicated to solve the query and adds it to the list of currently running query threads. When this thread reports back the result of the query, it is killed and removed from the list.

5.2 Query Servant

When a query servant thread is created, it is always associated with an ID and with the query containing the literal to be solved, and it is added to the list of running threads of the query component. In accordance with the $P2P_DR$ algorithm that we described in Section 3, the query servant model works as follows:

1. The first phase consists of trying to solve the query locally using the local knowledge base of the query component. If a positive/negative truth value is derived locally, the answer is returned and the query servant terminates.
2. The second phase consists of enumerating the rules in the knowledge base that support the queried literal as their conclusion. For each such rule, the query servant initiates a new query for each of the literals that are in the body

of the rule. For foreign literals, the queries are dispatched to the appropriate remote components. After initiating the queries, the query servant goes into an idle state through the java command "wait()".
3. When responses are received, the query servant thread is notified. Phase two is repeated again, but this time using the rules that support the negation of the queried literal.
4. The last step is to resolve the conflict by comparing the Supportive Sets of the rules that support the queried literals with the Supportive Sets of the rules that support its negation using the preference order. The result is reported back to the query component.

5.3 Query class and loop detection mechanism

The query Java class that we developed for our platform has the following attributes: the queried literal, the name of the component that initiated this query (*query owner*), the name of the component to which the query is addressed (*query recipient*), the id of the query servant thread that is responsible for solving this query, a set of supportive sets, and a list that represents the history of the query. The history is used to track back to the origin of the query by a loop detection mechanism, which we have integrated in $P2P_DR$.

As $P2P_DR$ is a distributed algorithm, we cannot know a-priori whether a query will initiate an infinite loop of messages. The loop detection mechanism that we developed detects and terminates any infinite loops. The simple case is when a literal $(c_i : a)$ in component C_i depends on literal $(c_k : b)$ of component C_k, and vice-versa. The loop detection mechanism works as follows: each time the query servant inquires about a foreign literal to solve the current query, it first checks that the foreign literal in question does not exist in the history of the current query, and if not, it generates a new query for the foreign literal by integrating the history of the current query into the history of the new one. This way, a query servant is only allowed to inquire about new literals.

5.4 Running example

Applying the above methodology on the running example described in section 2, we created 5 Query Component instances, each one representing one of the devices or elements of the scenario: the sms module, the bracelet, the medical profile, the ARM and the Home Care System. According to the scenario, the sms module must determine whether to send messages to the neighbors according to a predefined set of rules. Using a console component of Kevoree that we attached to the sms module, we were able to initiate queries on the sms module.

Figure 5 shows our experimental setup, which involves the 5 query components and the console component connected to the sms module (*FakeConsole*). Note that, all query input and output ports of the query components are connected to each other via the same message channel called *queryChannel*, to allow any component to communicate and send queries to any other component.

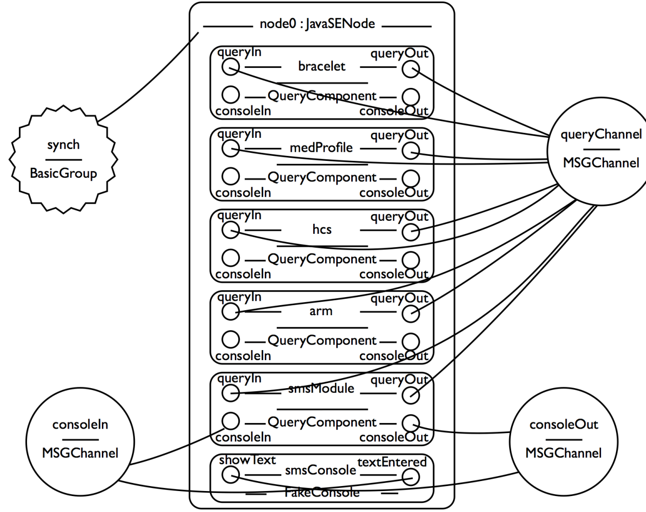

Fig. 5. The running example implemented, a snapshot of the Kevoree Editor.

File Name	**File contents**
smsModuleKB.txt	M1: (hcs:emergency) → (sms:dispatchSMS)
BraceletKB.txt	L1: → (br:normalPulse)
MedProfileKB.txt	L1: → (med:proneToHA)
ArmKB.txt	L1: → (arm:lyingOnFloor)
HCSKB.txt	M1: (br:normalPulse) ⇒ ¬(hcs:emergency) M2: (arm:lyingOnFloor), (med:proneToHA) ⇒ (has:emergency)
HCSPref.txt	med, arm, br

Table 1. Initialization of the components of the running example

Before pushing the model from the Kevoree editor to the Kevoree runtime (i.e.: the node that will host the instances), we setup the properties of the components to initialize their knowledge bases and preference orders as described in Table 1, and according to the CDL representation model that we present in Section 3. For instance, the *sms* component is initiated with a knowledge base containing one mapping rule ($M1$) that states that if ($hcs : emergency$) of *hcs* is true, then ($sms : dispatchSMS$) of the sms module will also be true. HCSPref.txt contains the preference order of *hcs*, according to which the information imported by the medical profile is preferred to that coming from the ARM, which is in turn preferred to that coming from the bracelet.

After pushing the model to the Kevoree runtime, a console appears allowing us to interact with the sms module. We initiate a query about ($sms : dispatchSMS$), and we get *true* as a response. In fact, what happens in the back-end is that a query servant starts on the sms module to solve the query. The query servant initiates, then, a new query for ($hcs : emergency$). In the

knowledge base of *hcs*, there is one rule supporting this literal, and another one supporting its negation. *hcs* evaluates both rules and resolves the conflict using its preference order. Finally, it sends back the result of the query to the first query servant, which in turn computes and returns a positive truth value for $(sms : dispatchSMS)$. The full interaction is displayed in figure 6.

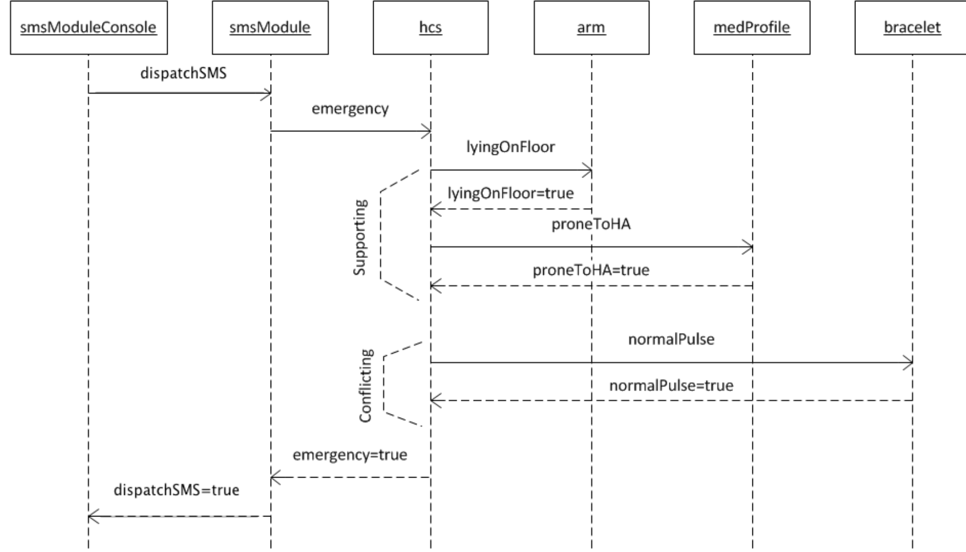

Fig. 6. Execution of the running example.

5.5 Limitations

Our platform still has some technical limitations. As it deals with real components, we must assume limited memory, battery, computation and power resources. These limitations vary widely from a component to another depending on the nature of the component, its size and its technical complexity. For the current implementation, we have limited the knowledge base size to a maximum of 500 literals and rules. We have also limited the time-out for 10 seconds, so that if a component does not receive an answer to its query within 10 seconds, the corresponding thread server will send a time-out response, and the query will automatically expire. This limits the maximum number of hops that a query can make before it expires, which in turn limits the communication resources, as some communication channel might not be free (over sms for example). With the current settings, we can easily implement small-scale AAL scenarios. However, dealing with more complex scenarios requires a more scalable methodology. To address such needs, we are already working on solutions that offer trade-offs between computation time, memory and communication between devices, and we are redesigning our algorithms so that they able to adapt between different strategies depending on the available resources.

6 Related Work

Rule-based approaches offer several benefits with respect to reasoning about context in Ambient Intelligence environments, such as *simplicity and flexibility*, *formality*, *expressivity*, *modularity*, *high-level abstraction* and *information hiding* [14]. Various logics have been proposed so far for such purposes including: First Order Logic [15, 16], Logic Programming [17, 18], Answer Set Programming [19] and Defeasible Logic [20]. Classical reasoning approaches (e.g. First Order Logic) are based on the assumption of perfect knowledge of context, which, as we explained in Section 1, is not valid in Ambient Intelligence environments. Non-monotonic approaches, including the one that we propose in this paper, enable reasoning with imperfect context, adding though additional complexity overhead to the reasoning process. Although the complexity of the algorithm that we use for query evaluation is exponential [3], it is among our future plans to design new algorithms that will exploit the linear complexity of Defeasible Logic.

Although Ambient Intelligence environments are distributed by nature, all systems that are cited above are based on centralized architectures: a central entity is responsible for collecting relevant context data from all sensors and devices operating in the same environment, and for conducting the contextual reasoning tasks. The *shared memory* and *blackboard* models that were used by other systems (e.g [21–23]) are also based on the assumption of a central place where all relevant data is collected and processed. However, in such environments context changes may be very frequent, devices may join or disconnect at random times and without prior notice, while wireless communications are unreliable and restricted by the range of the transmitters. Moreover, privacy restrictions may be applied by the users, according to which part of the data stored in a device must remain local. Therefore, a totally distributed model, such as the one that we propose here, fits better with such requirements and needs.

7 Conclusion

In this paper, we address some of the challenges imposed by the special characteristics and requirements of intelligent environments to the reasoning processes of Ambient Intelligence systems. Such systems must enable heterogeneous entities, which operate in open and dynamic environments, to collectively reason with imperfect context information. We build on previous work, in which we introduced Contextual Defeasible Logic as a contextual reasoning model to address most of these challenges using the concepts of *context*, *mappings* and *contextual preferences*. In this paper, we first introduce the implementation of the logic in Kevoree, a component-based software platform for Distributed Dynamically Adaptive Systems. Second, we present an Ambient Assisted Living (AAL) scenario, which we use as a running example to present the main aspects of our platform. We describe how we implemented the reasoning model of CDL in Kevoree, and explain how the capabilities of Kevoree are exploited to overcome several technical issues, such as communication, information exchange and

detection. Third, we discuss the additional technical issues that arise from the deployment of CDL in real environments, and propose ways to resolve them. Finally, we emphasize that we provide a platform, which anyone may use to test and implement scenarios from any field of Ambient Intelligence.

In the future, we plan to extend CDL to support shared pieces of knowledge, which are directly accessible by all system contexts, and implement this extension in Kevoree using its *groups* feature (see section 4). This will enable different devices operating in an Ambient Intelligence environment to maintain a common system state. We also plan to develop and implement reactive (bottom-up) reasoning algorithms, which will be triggered by certain events or changes in the environment. Such types of algorithms fit better with the adaptive nature of Ambient Intelligence systems, and may be particularly useful in AAL contexts. We will also study the integration of a low-level context layer in our platform, which will process the available sensor data and feed the rule-based reasoning algorithms with appropriate values for the higher-level predicates. For this layer, we will investigate the Complex Event Processing (*CEP*) methodology [24], which combines data from multiple sources to infer higher-level conclusions, and we will build on top of previous works that study the integration of CEP and reaction rules [25]. We will test and evaluate all our deployments and extensions to our platform in the Internet of Things Laboratory of the Interdisciplinary Centre for Security, Reliability and Trust (SnT) in Luxembourg. It is also among our plans to use our platform to evaluate tradeoffs among requirements of AAL systems, e.g., privacy, security, usability/conviviality and performance. Finally, we plan to investigate how the same reasoning methods may be applied to other application areas with similar requirements, such as the Semantic Web and Web Social Networks.

References

1. Cook, D.J., Augusto, J.C., Jakkula, V.R.: Ambient intelligence: Technologies, applications, and opportunities. Pervasive and Mobile Computing (2009) 277–298
2. Henricksen, K., Indulska, J.: Modelling and Using Imperfect Context Information. In: Proceedings of PERCOMW '04, Washington, DC, USA, IEEE Computer Society (2004) 33–37
3. Bikakis, A., Antoniou, G.: Defeasible Contextual Reasoning with Arguments in Ambient Intelligence. IEEE Trans. on Knowledge and Data Engineering **22**(11) (2010) 1492–1506
4. Bikakis, A., Antoniou, G., Hassapis, P.: Strategies for contextual reasoning with conflicts in Ambient Intelligence. Knowledge and Information Systems **27**(1) (2011) 45–84
5. Moawad, A., Efthymiou, V., Caire, P., Nain, G., Le Traon, Y.: Introducing conviviality as a new paradigm for interactions among IT objects. In: Proceedings of the Workshop on AI Problems and Approaches for Intelligent Environments. Volume 907., CEUR-WS.org (2012) 3–8
6. Efthymiou, V., Caire, P., Bikakis, A.: Modeling and evaluating cooperation in multi-context systems using conviviality. In: Proceedings of BNAIC 2012 The 24th Benelux Conference on Artificial Intelligence. (2012) 83–90

7. Fouquet, F., Barais, O., Plouzeau, N., Jézéquel, J.M., Morin, B., Fleurey, F.: A Dynamic Component Model for Cyber Physical Systems. In: 15th International ACM SIGSOFT Symposium on Component Based Software Engineering, Bertinoro, Italie (July 2012)
8. Antoniou, G., Billington, D., Governatori, G., Maher, M.J.: Representation results for defeasible logic. ACM Transactions on Computational Logic **2**(2) (2001) 255–287
9. Giunchiglia, F., Serafini, L.: Multilanguage hierarchical logics, or: how we can do without modal logics. Artificial Intelligence **65**(1) (1994)
10. Ghidini, C., Giunchiglia, F.: Local Models Semantics, or contextual reasoning=locality+compatibility. Artificial Intelligence **127**(2) (2001) 221–259
11. Bikakis, A., Antoniou, G.: Partial preferences and ambiguity resolution in contextual defeasible logic. In: LPNMR. (2011) 193–198
12. Morin, B., Barais, O., Nain, G., Jezequel, J.M.: Taming dynamically adaptive systems using models and aspects. In: Proceedings of the 31st International Conference on Software Engineering. ICSE '09, Washington, DC, USA, IEEE Computer Society (2009) 122–132
13. Fouquet, F., Nain, G., Morin, B., Daubert, E., Barais, O., Plouzeau, N., Jézéquel, J.M.: An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements. In: Models 2012, Innsbruck, Autriche (October 2012)
14. Bikakis, A., Antoniou, G.: Rule-based contextual reasoning in ambient intelligence. In: RuleML. (2010) 74–88
15. Ranganathan, A., Campbell, R.H.: An infrastructure for context-awareness based on first order logic. Personal Ubiquitous Comput. **7**(6) (2003) 353–364
16. Gu, T., Pung, H.K., Zhang, D.Q.: A Middleware for Building Context-Aware Mobile Services. In: Proceedings of the IEEE Vehicular Technology Conference (VTC 2004), Milan, Italy (May 2004)
17. Toninelli, A., Montanari, R., Kagal, L., Lassila, O.: A Semantic Context-Aware Access Control Framework for Secure Collaborations in Pervasive Computing Environments. In: Proc. of 5th International Semantic Web Conference. (November 2006) 5–9
18. Agostini, A., Bettini, C., Riboni, D.: Experience Report: Ontological Reasoning for Context-aware Internet Services. In: PERCOMW '06: Proceedings of the 4th annual IEEE international conference on Pervasive Computing and Communications Workshops, Washington, DC, USA, IEEE Computer Society (2006)
19. Mileo, A., Merico, D., Pinardi, S., Bisiani, R.: A logical approach to home healthcare with intelligent sensor-network support. Comput. J. **53**(8) (2010) 1257–1276
20. Antoniou, G., Bikakis, A., Karamolegou, A., Papachristodoulou, N., Stratakis, M.: A context-aware meeting alert using semantic web and rule technology. International Journal of Metadata Semantics and Ontologies **2**(3) (2007) 147–156
21. Khushraj, D., Lassila, O., Finin, T.: sTuples: Semantic Tuple Spaces. In: First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous04). (August 2004) 267–277
22. Krummenacher, R., Kopecký, J., Strang, T.: Sharing Context Information in Semantic Spaces. In: OTM Workshops. (2005) 229–232
23. Korpipaa, P., Mantyjarvi, J., Kela, J., Keranen, H., Malm, E.J.: Managing Context Information in Mobile Devices. IEEE Pervasive Computing **02**(3) (2003) 42–51
24. Luckham, D.C.: The power of events - an introduction to complex event processing in distributed enterprise systems. ACM (2005)
25. Paschke, A., Vincent, P., Springer, F.: Standards for complex event processing and reaction rules. In: RuleML America. (2011) 128–139