R-CoRe: A Rule-based Contextual Reasoning Platform for AmI ⋆

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. In this paper we present R-CoRe; a rule-based contextual reasoning platform for Ambient Intelligence environments. R-CoRe integrates Contextual Defeasible Logic (CDL) and Kevoree, a component-based software platform for Dynamically Adaptive Systems. Previously, we explained how this integration enables to overcome several reasoning and technical issues that arise from the imperfect nature of context knowledge, the open and dynamic nature of Ambient Intelligence environments, and the restrictions of wireless communications. Here, we focus more on technical aspects related to the architecture of R-Core, and demonstrate its use in Ambient Assisted Living.

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

1 Introduction

Ambient Intelligence (AmI) is 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 [6]. Therefore, they must be able to reason about their *contexts*, i.e. with any information relevant to the interactions between the users and system.

Reasoning models and methods are therefore essential to: interpret and integrate context data from various information sources; infer useful conclusions from the raw context data; and enable systems to make correct context-aware decisions in order to adapt to changes in the environment and to their users' needs, intentions and desires. The challenges in these tasks are primarily caused by the

⋆ The present research is supported by the National Research Fund, Luxembourg, CoPAInS project (code: CO11/IS/1239572).

imperfection of context data, the open and dynamic nature of AmI environments and the heterogeneity of participating devices. According to [12], context data in AmI environments may be unknown, imprecise, ambiguous or erroneous. It is typically distributed among devices with different computing capabilities and representation models, which may join or leave the environment at random times and without prior notice. Moreover, devices communicate using wireless networks, which are unreliable and restricted by the range of transmitters.

In previous works we classified existing contextual reasoning approaches into three main categories [3]: *ontological* approaches, which use Description Logics to derive implicit knowledge from the existing context data; *rule-based* approaches, which are based on more expressive logics; and *probabilistic* approaches, which use Bayesian networks or other probabilistic models to explicitly model uncertainty in the context data. In the same paper we argued that, compared to others, rule-based approaches offer significant advantages, such as *simplicity* and *flexibility*, *formality*, *expressivity*, *modularity*, *high-level abstraction* and *information hiding.* In [2] we introduced *Contextual Defeasible Logic* (CDL): a new distributed, non-monotonic approach for contextual reasoning, which enables heterogeneous entities to collectively reason with uncertain and ambiguous information. In order to enable the deployment of CDL in real environments, in [15] we proposed the integration of CDL in Kevoree [8] - a software framework that facilitates the development of Distributed Dynamically Adaptive Systems.

In this paper, we present R-CoRe, a Rule-based Contextual Reasoning Platform of Ambient Intelligence, which is the outcome of this integration. The main features of R-CoRe are:

1. It is totally distributed. Entities are represented as Kevoree nodes and communicate through dedicated communication channels.
2. It is rule-based. The local knowledge of each entity is modeled as a CDL theory and knowledge exchange is enabled by mapping rules.
3. It enables handling inconsistencies that arise from the integration of knowledge from different sources using preferences, which reflect the confidence that each node has in the quality of knowledge imported by other nodes.
4. It is dynamic and adaptive. Using the auto-discovery capabilities of Kevoree, it can handle cases of devices that join or leave the system at any time.

We developed R-CoRe for the needs of the CoPAInS project[1] (Conviviality and Privacy in Ambient Intelligence Systems). CoPAInS focuses on the tradeoffs to be made in Ambient Assisted Living systems [16], particularly as they pertain to conviviality, privacy and security [7]. In this framework, R-CoRe is used for the simulation of AAL scenarios, with which we test and validate our methods.

The remaining of the paper is structured as follows: Section 2 briefly presents the theoretical background of this work. Section 3 describes our running AAL example. Section 4 presents the main features of Kevoree. Section 5 presents in detail the R-CoRe architecture, while Section 6 demonstrates its use in Ambient Assisted Living. Section 7 concludes and presents our plans for future work.

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

2 Background

The underlying reasoning model of R-CoRe is Contextual Defeasible Logic (CDL [2,5]). CDL is a non-monotonic extension of Multi-Context Systems specifically designed for the requirements of Ambient Intelligence systems. Multi-Context Systems (MCS [11,10]) are 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. CDL extends the original MCS with defeasible mapping rules to capture the uncertainty of the knowledge that an agent imports from external sources; and with a preference ordering on the system contexts, which is used to resolve the potential inconsistencies that may arise from the information exchange between mutually inconsistent 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 (a set of positive and negative literals of the form $(c_i : a_i)$), R_i is a set of rules, and T_i is a preference ordering on C. R_i consists of a set of *local rules*, which represent the local knowledge of an agent, and a set of *mapping rules*, through which agents may share parts of their local knowledge. The body of a local rule is a conjunction of *local* literals (literals that are contained in V_i), and its head is labeled by a local literal too. A mapping rule, on the other hand, contains both local and foreign literals (literals from the vocabularies of other contexts) in its body, while its head is labeled by a local literal:

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

By representing mappings as defeasible rules and by ordering contexts in terms of preference, CDL enables handling inconsistencies that arise when importing conflicting information from different contexts.

We have obtained the following results for CDL: a proof theory [4]; an argumentation semantics [2]; four algorithms for distributed query evaluation and a complexity analysis [5]. The algorithms proceed roughly as follows: when a context C_i receives a query about one of its local literals $(c_i : a_i)$, it first attempts to evaluate its truth value using its local rules only. If this is not possible, it generates the proof trees for $(c_i : a_i)$ and its negation $\neg(c_i : a_i)$, using the local and mapping rules of C_i. For each of the foreign literals $(c_j : b_j)$ contained in one of the two proof trees, it generates a similar query and sends it to the appropriate context (C_j). In case of conflict, i.e.. the truth values of all literals in both proof trees are true, C_i compares the proof trees using preference information from T_i.

We applied CDL in real scenarios of Mobile Social Networks [1] and Ambient Intelligence [5], and showed how it addresses issues that arise from the imperfection of context data. In [15], we discussed some of its limitations with regard to its deployment in real environments, and explained how its integration with Kevoree enabled us to overcome several technical issues related to communication, detection, adaptability and dynamicity. Here we focus more on technical aspects of this integration, and demonstrate the use of the integrated framework, R-CoRe, in an example scenario from Ambient Assisted Living.

3 An Ambient Assisted Living Example

Below 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 the CoPAInS project.

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.

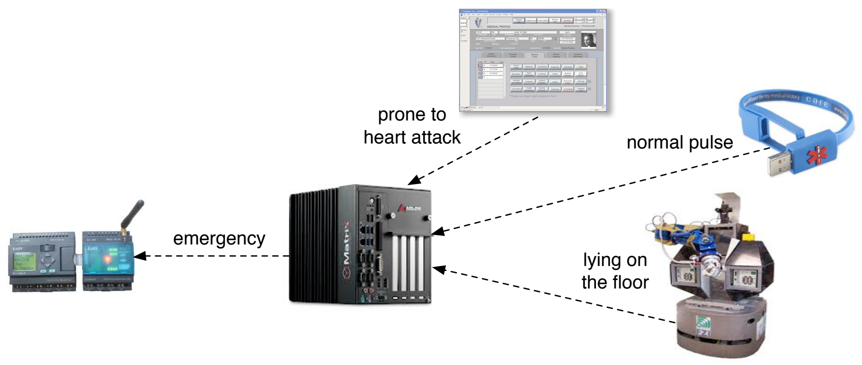

Fig. 1. Context Information flow in the scenario.

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.

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.

AAL
AMBIENT ASSISTED LIVING
Kevoree
CDL

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

4.1 Kevoree: Modeling Framework and Components

Kevoree [8] is an open-source environment that provides means to facilitate the design and deployment of Distributed Dynamically Adaptive Systems, taking advantage of Models@Runtime [17] mechanisms throughout the development process.

This development platform is made of several tools, among which the Kevoree Modeling Framework (KMF) [9], 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.

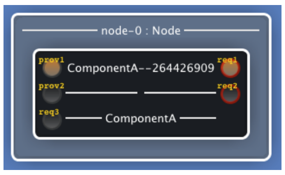

Fig. 3. A component instance, inside the node instance on which it executes

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.

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.

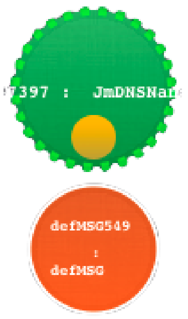

Fig. 4. An instance of Group on top, of Channel on the bottom

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 R-CoRe Architecture

In this section, we describe how CDL and Kevoree are integrated in the R-CoRe architecture. We should note that the parts of CDL, which were not directly mapped to existing elements Kevoree, were implemented in Java.

5.1 Java Library

Our Java implementation is composed of a main rcore package containing 4 sub-packages: agencies, interceptor, knowledge, logic packages and the main Query component class.

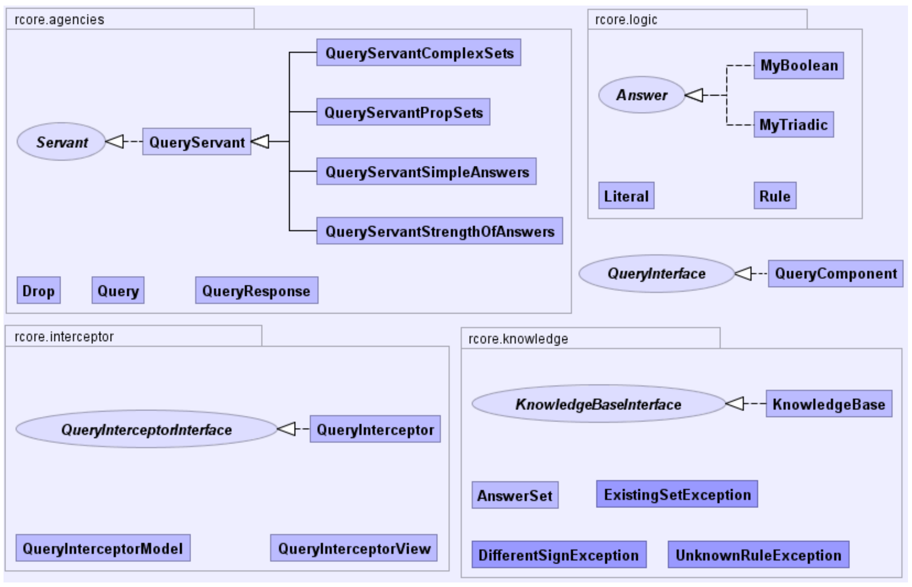

Fig. 5. General overview of R-CoRe.

– The agencies package contains the classes used by the query servant thread, which implements the reasoning algorithms of CDL [5]. This package contains also the Query class and the QueryResponse class. These classes represent the massages that will be exchanged between QueryComponents.
– The interceptor package contains the model, the view and the controller of the interceptor component. The controller is a component developed to run on the Kevoree platform.
– The knowledge package includes the KnowledgeBase class, which stores the local rule theory, the mapping rules and the preference order of each node. This class contains also the methods used to load and save the knowledge base and preference order from and to files.

– The logic package contains the classes that represent (in memory) the literals and rules.
– Finally the query package contains the QueryComponent class, which is the main component developed to run on the Kevoree platform.

Figure 5 shows a UML overview of our implementation.

5.2 Query Component

In our platform, the notion of context, 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 R-CoRe, 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.

5.3 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. 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 remote components that were queried for the two literals using the context preference order. The result is reported back to the query component.

5.4 Query Interceptor

In order to monitor and control all the exchanges happening between the Query components, we created a Query Interceptor component. It’s main job is to capture all the queries transiting on the exchange channel, display them on a graphical interface to the users, and allow the users to forward them manually afterwards. This component serves two purposes: It enables demonstrating the reasoning process using a single graph that is created in a step by step fashion; it also facilitates debugging the system by centralizing all the exchanges in one component.

The Graphical interface of the Query Interceptor has two parts: the graph on the left that is used to visualize the information exchange; and the user controls on the right. When a query is sent from a component a to a component b through the interceptor, two vertices, a and b, and an edge from a to b are added to the graph. If the Interceptor is set to the *demo-mode* (by selecting a check box called ”demo mode“ on the Graphical User Interface GUI), the query is paused at the interceptor, and the user has to click the *Next* button in order to actually forward the query from the interceptor to component b. The same happens when a response is sent from b to a. If the *demo-mode* is not selected, all the queries and responses are forwarded automatically without any intervention of the user as if the Interceptor is transparent or turned off. The Interceptor also contains *Reset* button, which clears the graph and restarts the monitoring.

On the Kevoree platform, the Query Interceptor component has two ports: *Query In* port that receives all the queries sent from the Query components, and a *Query out* port to forward the query back to the components after being displayed on the Interceptor GUI.

5.5 Query class and loop detection mechanism

The query Java class that we developed for R-CoRe has the following attributes: the queried literal, the name of the component that initiated the 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 evaluating the query, a set of supportive sets (set of foreign literals that are used for the evaluation of a query), 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 the query evaluation algorithm.

As the query evaluation algorithm is distributed, 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.

6 Demonstrating R-Core

6.1 Setup

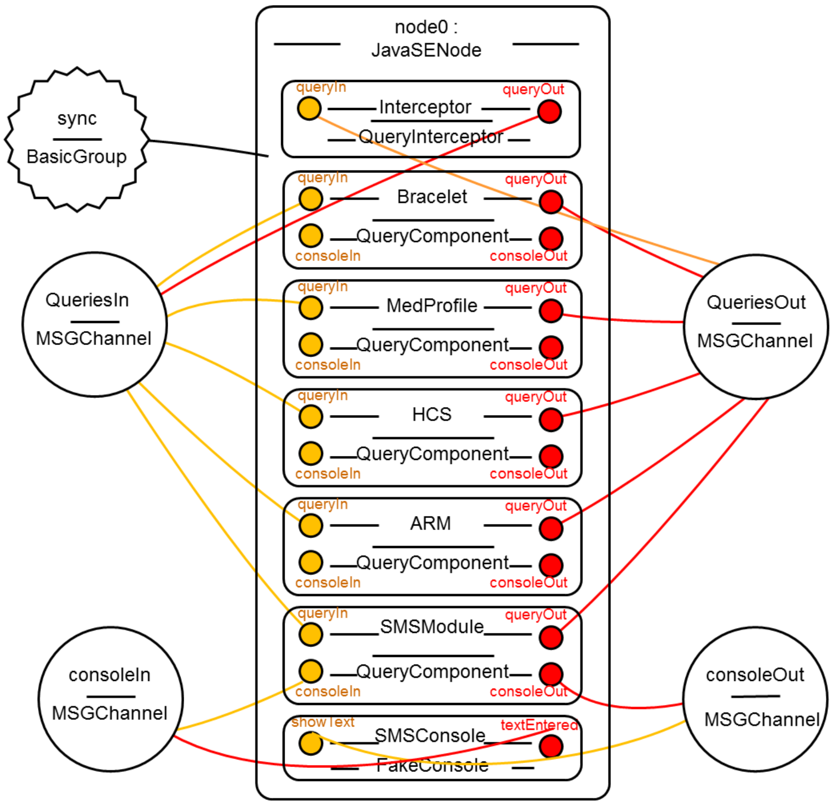

Fig. 6. The running example implemented, a snapshot of the Kevoree Editor.

Applying the above methodology on the running example described in section 3, 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 are able now to initiate queries on the sms module.

Figure 6 shows our experimental setup, which involves the 5 query components, the console connected to the sms module (*FakeConsole*) and the Query Interceptor component. Note that all query input and output ports of the query components are connected to the Interceptor in order to allow us to capture all the exchanges for our demo session.

Table 1. Initialization of the components of the running example

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

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. 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.

6.2 Execution

After pushing the model to the Kevoree runtime, a console appears allowing us to interact with the SMS module. We initiate a query about ($sms : dispatchSMS$) on the console by typing ***dispatchSMS*** on the console of the SMSModule.[2]

The SMS module starts a new query servant which initiates in its turn a new query about ($hcs : emergency$). This query is captured by the interceptor and it

[2] You can run the demo and access its source code and all other necessary files at: `https://github.com/securityandtrust/ruleml13`.

is displayed on the graph GUI; in fact two nodes representing the SMSModule and HCS are added to the graph. If the *demo mode* of the interceptor is selected, the user has to click on *Next* button each time to forward a query from one component to another. This step-by-step mode is very useful to understand the actual interactions and to slow down the exchanges.

The knowledge base of hcs contains one rule supporting ($hcs : emergency$), $M2$, and another one supporting its negation, $M1$. 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 ($sms : dispatchSMS$). Each time a query or a query response is generated from any component and dispatched to any other, the interceptor captures it, adds it to the graph, and waits for the user to click Next before forwarding the query or the response to the appropriate component.

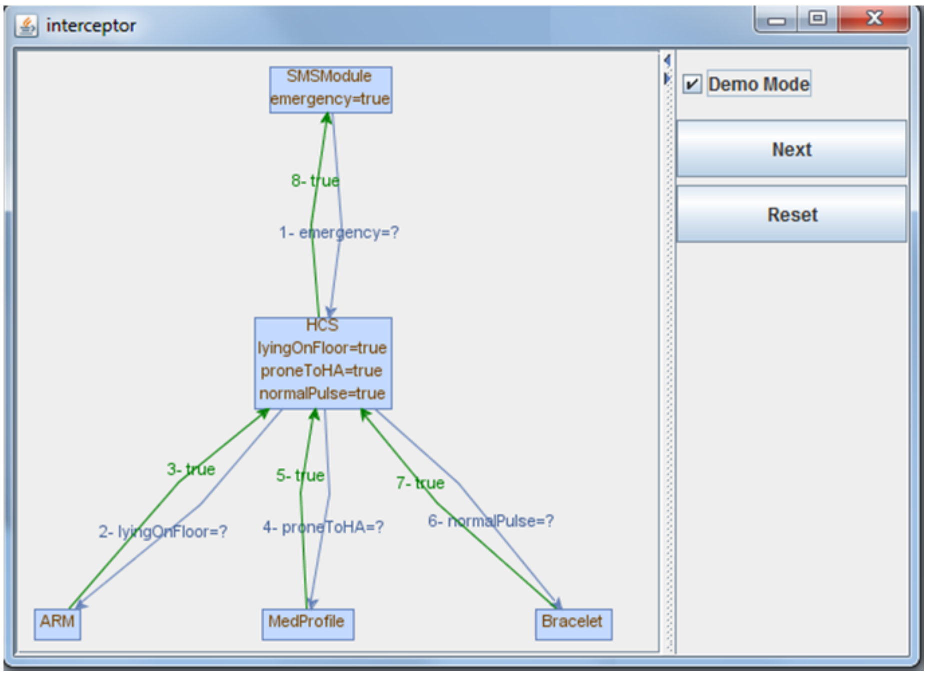

Fig. 7. Different steps of execution of the running example.

Screen-shots from the demo during different steps of execution are displayed in figure 7. Following this, the SMS module will display, on the console connected to smsModule, the answer for $dispatchSMS$, which in this case is $true$.

7 Conclusion and Future Work

In this paper, we presented R-CoRe: a Rule-based Contextual Reasoning framework for Ambient Intelligence and demonstrated its use Ambient Assisted Living. Being based on Contextual Defeasible Logic, R-CoRe enables reasoning with imperfect context data in a distributed fashion. The capabilities of the underlying software platform, Kevoree, further enables R-CoRe to overcome several technical issues related to communication, information exchange and detection.

R-CoRe 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 are able to adapt between different strategies depending on the available resources.

Another issue is with the development of the rule theories. In this version of R-CoRe, users have to use the syntax of CDL to create the rule and preference bases of each node. A future plan is to develop or integrate appropriate rule-editing tools that will enable plain users to create and configure the rule and preference bases using simple natural-language-based constructs. A tool that we can use for such purposes is S^{2}DDREd [13]: an authoring tool for Defeasible Logic, which provides the users with semantic assistance during the development of rule theories.

In the future, we also plan to extend CDL to support shared pieces of knowledge, which are directly accessible by all system contexts, and implement this extension in R-CoRe using the *groups* feature of Kevoree (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 R-CoRe, 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 [14], which combines data from multiple sources to infer higher-level conclusions, and study previous works on the integration of CEP and reaction rules [18]. We will test and evaluate all our deployments and extensions to R-CoRe 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. Antoniou, G., Papatheodorou, C., Bikakis, A.: Reasoning about Context in Ambient Intelligence Environments: A Report from the Field. In: KR. pp. 557–559. AAAI Press (2010)
2. Bikakis, A., Antoniou, G.: Defeasible Contextual Reasoning with Arguments in Ambient Intelligence. IEEE Trans. on Knowledge and Data Engineering 22(11), 1492–1506 (2010)
3. Bikakis, A., Antoniou, G.: Rule-based contextual reasoning in ambient intelligence. In: RuleML. pp. 74–88 (2010)
4. Bikakis, A., Antoniou, G.: Contextual Defeasible Logic and Its Application to Ambient Intelligence. IEEE Transactions on Systems, Man, and Cybernetics, Part A 41(4), 705–716 (2011)
5. Bikakis, A., Antoniou, G., Hassapis, P.: Strategies for contextual reasoning with conflicts in Ambient Intelligence. Knowledge and Information Systems 27(1), 45–84 (2011)
6. Cook, D.J., Augusto, J.C., Jakkula, V.R.: Ambient intelligence: Technologies, applications, and opportunities. Pervasive and Mobile Computing pp. 277–298 (2009)
7. 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. pp. 83–90 (2012)
8. 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 (Jul 2012), http://hal.inria.fr/hal-00713769
9. 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 (Oct 2012), http://hal.inria.fr/hal-00714558
10. Ghidini, C., Giunchiglia, F.: Local Models Semantics, or contextual reasoning=locality+compatibility. Artificial Intelligence 127(2), 221–259 (2001)
11. Giunchiglia, F., Serafini, L.: Multilanguage hierarchical logics, or: how we can do without modal logics. Artificial Intelligence 65(1) (1994)
12. Henricksen, K., Indulska, J.: Modelling and Using Imperfect Context Information. In: Proceedings of PERCOMW '04. pp. 33–37. IEEE Computer Society, Washington, DC, USA (2004)
13. Kontopoulos, E., Zetta, T., Bassiliades, N.: Semantically-enhanced authoring of defeasible logic rule bases in the semantic web. In: WIMS. p. 56 (2012)
14. Luckham, D.C.: The power of events - an introduction to complex event processing in distributed enterprise systems. ACM (2005)

15. Moawad, A., Bikakis, A., Caire, P., Nain, G., Traon, Y.L.: A Rule-based Contextual Reasoning Platform for Ambient Intelligence environments. In: RuleML. LNCS, Springer (2013)
16. 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. vol. 907, pp. 3–8. CEUR-WS.org (2012)
17. 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. pp. 122–132. ICSE '09, IEEE Computer Society, Washington, DC, USA (2009), `http://dx.doi.org/10.1109/ICSE.2009.5070514`
18. Paschke, A., Vincent, P., Springer, F.: Standards for complex event processing and reaction rules. In: RuleML America. pp. 128–139 (2011)