Evaluation of a Rule-Based Approach for Context-Aware Services

Patricia Dockhorn Costa and João Paulo A. Almeida
Department of Computer Science
Federal University of Espírito Santo (UFES)
Vitoria, Brazil
{pdcosta, jpalmeida}@inf.ufes.br

Luís Ferreira Pires and Marten van Sinderen
Centre for Telematics and Information Technology
University of Twente
Enschede, the Netherlands
{l.ferreirapires, m.j.vansinderen}@ewi.utwente.nl

***Abstract*—In previous work we have proposed a rule-based platform to support context-aware service development. Rules are used to detect relevant situations in the users' context and to trigger actions in response to context changes. Rule-based solutions offer flexibility with respect to service maintainability, since rules can be modified and added at runtime. However, rule-based systems are often criticized for being monolithic systems that only perform adequately for rather static datasets, which would imply that these systems are not capable of providing the performance and scalability required by large scale context-aware services. In this paper we show that this criticism does not always hold by evaluating our distributed rule-based implementation and demonstrating the suitability of this approach for highly dynamic context-aware services.**

***Index Terms*—context-awareness, rule-based systems**

I. INTRODUCTION

Context-aware services use and manipulate context information to detect the situations of their users and adapt service behaviour accordingly. These services not only use context information to react on a user's request, but also take initiative as a result of context reasoning activities. In this sense, such services can be characterized as *attentive* in addition to *reactive*. Attentive context-aware services must continuously monitor the user's environment in order to detect changes and react to them, which is a capability that imposes strict performance and scalability constraints on their possible realizations. These constraints are particularly stringent for a large number of users and context variables.

In previous work [3][4] we have addressed the development of context-aware services through a context handling framework. We have proposed a context handling platform to allow service functionality to be delegated to the platform, reducing development effort. The delegated functionality detects the occurrence of relevant context situations, in order to notify service components or perform actions on their behalf.

In our approach, context situations are detected using a rule-based realization. We use a mature off-the-shelf rule engine (Jess, [5][10]), which runs rule sets that are systematically derived from context-aware service specifications.

Rule-based solutions offer flexibility with respect to service maintainability, since rules can be modified and added at runtime. However, rule-based systems are often criticized for being monolithic systems that only perform adequately for rather static datasets [11], which would imply that these systems are not capable of providing the performance and scalability required by large scale context-aware services. In this paper we show that this criticism does not always hold, by evaluating our distributed rule-based implementation and demonstrating the suitability of this approach for highly dynamic context-aware services.

In order to evaluate the performance of our context-aware platform, we introduce a *reaction time* parameter, which indicates the period of time between generating relevant events and invoking an action. Our experiments show that a context-aware healthcare service built with our platform performs satisfactory, i.e., that the reaction time of this service is acceptable even for a large number of users and events.

This paper is further structured as follows: section II provides some background by introducing our rule-based approach from previous work [3][4], section III discusses the healthcare service design, section IV presents the prototype that implements this design; section V discusses our results, by providing a quantitative evaluation of our approach in the healthcare scenario, and, finally, section VI presents some concluding remarks.

II. RULE-BASED APPROACH

The context handling platform we have proposed is based on the Event-Control-Action (ECA) pattern discussed in [2]. This architectural pattern aims at providing a structural scheme to enable the coordination, configuration and cooperation of distributed functionality in a context-aware platform. The ECA pattern separates the tasks of gathering and processing context information from the tasks of triggering actions in response to context changes.

This work is part of the Freeband AWARENESS and A-MUSE projects (http://awareness.freeband.nl and http://amuse.freedband.nl). Freeband is sponsored by the Dutch government under contract BSIK 03025. This work is also partly supported by FAPES (Fundação de Apoio à Ciência e Tecnologia do Espírito Santo) and CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) in the scope of the INFRA-MODELA project.

With this pattern, context-aware service behaviours can be described as logic rules, which are called Event-Condition-Action (ECA) rules. In these rules, Events model occurrences of interest (e.g., changes in context); Conditions specify the conditions that must hold prior the execution of actions; and Actions represent the invocation of arbitrary services. We have developed a domain specific language called ECA-DL for the purpose of specifying ECA rules. By means of this language, services developers are capable of specifying rules that express desired context-aware reactive behaviours in a scripting format, which can be deployed at platform runtime.

In order to carry out control on behalf of services according to the ECA pattern, we have proposed a controller component as part of the context-handling platform. The controller component receives service-specific ECA-DL specifications as input, and executes these behaviours in the platform on behalf of services. In order to execute these service-specific behaviours, the controller observes event notification asynchronously, monitors condition rules, and triggers actions when particular events occur and conditions are satisfied.

The platform supports three kinds of events: (i) primitive events; (ii) temporal events and (iii) situation events. Primitive and temporal events require no reasoning on the user's context. Situation events, in contrast, are detected by continuously monitoring the context. Since we are interested in the evaluation of reasoning activities in the rule-based approach, we focus here on the detection of situation events (the most challenging case from the perspective of performance).

Situations are particular state-of-affairs which are of interest of the context-aware service. Examples of situations that may be of interest to a context-aware service are "user is running and he/she has access to his/her mobile phone", and "user is in danger of an eminent epileptic seizure and he/she is driving".

Situations are defined by using a composition of elements, which include context conditions, temporal clauses and general constraints on entities, objects and their attributes. For this reason, situation specification can be potentially complex, requiring adequate modeling techniques. In our approach, a wide range of situation types can be specified using standard UML 2.0 [8] class diagrams, and OCL 2.0 [7] constraints to define the conditions under which situations of a certain type exist.

Situation detection is a continuously-running activity which gathers and processes continuously-changing context data from several distributed context sources and delivers situation event notifications to a multitude of consumers (for example ECA-DL rules). In our approach, situation detection is realized through a rule-based approach using the Jess rule engine.

In order to transform situation specifications into sets of rules to be executed directly on a Jess engine, we identify a number of patterns for rule detection realization [2]. Rules are systematically derived from UML and OCL specifications and deployed directly on the Jess rule engine [5].

Similarly to the detection of situations, the core of the realization of the controller is based on the Jess rule engine upon which rules corresponding to the ECA-DL specifications are executed. A mapping from ECA-DL specifications to Jess rules as well as ECA-DL syntax and semantics are beyond the scope of this paper and are fully elaborated in [2]. Since ECA-DL manipulates complex compositions of events, which includes temporal event relations, we have also provided means to cope with the lack of event support in Jess, without any extensions to the Jess core.

Fig. 1 shows a particular configuration of rule engines working together in a context-aware service. In Fig. 1, two rule engines perform situation detection, generating situation events (on situation detection) to a controller that executes the ECA-DL rules on behalf of the service, triggering action services when necessary. The mechanism used for rule execution (and in this case situation detection) is based on the Rete algorithm [5], which efficiently matches the patterns for situations by remembering past pattern matching tests. Only new or modified facts in the working memory are tested against the rules.

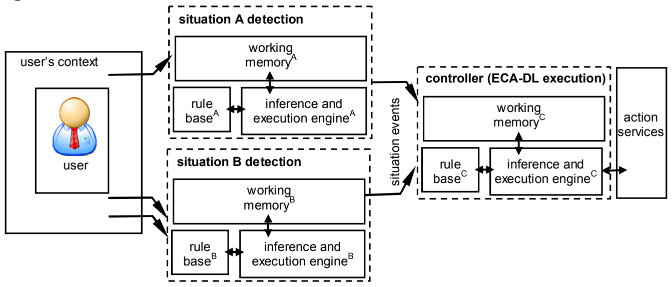

Figure 1 A configuration of a context-aware service realization

III. The Healthcare Telemonitoring Service

The development of our healthcare example is briefly discussed in this section, in terms of the usage scenario, the service design and the ECA-DL rules for this service.

A. *Usage Scenario*

The aim of the healthcare telemonitoring service is to improve the quality of life of epileptic patients through remote monitoring. This scenario has been used in the AWARENESS project [1], and its relevance for improving the quality of life of such patients has been confirmed by Roessingh Research and Development [9].

The service detects seizures and notifies both the epileptic patient and his/her nearby caregivers. The caregivers who receive the notification and patient location map should be: (i) assigned as one of the caregivers of that particular patient; (ii) available for helping; and (iii) physically close to the patient. Upon a notification for help, caregivers may either accept or reject the request for helping the epileptic patient.

B. *Service Design*

According to our development approach, the first step towards service realization is the modelling of the service's universe of discourse, emphasizing context conditions and situations. Our methodology for context and situation modelling has been discussed in [3], which also presents a complete elaboration of the modelling phase for this scenario.

Following context modelling, service design continues by identifying rules that can be delegated to the platform. At this point, we identify the context processor components (context sources and managers), which are needed for capturing the relevant context and situation information types.

The service components capable of offering epileptic alarms, patient's activity information, caregiver status information and caregiver's acceptance or rejection for help notifications, play the role of *context sources*, i.e., they obtain context information and events from the user's environment directly. The following context sources are required to support our scenario: EpilepticAlarmCS, GeoLocationCS, HazardousActivityCS, CaregiverStatusCS, and AcceptRequestCS.

We define two situation types of interest to the epileptic scenario, namely SituationCaregiverAvailable, and SituationCaregiver-WithinRange. Situation type SituationCaregiverAvailable specifies that caregivers are available when their status is set to "onCall" or "emergencyOnly". Situation type SituationCaregiverWithinRange specifies that a patient and a caregiver are within 100 meters distance from each other. These situations are detected by context manager components connected to context sources that enable them to detect these situations. Given the situation information requirements, we conclude that context managers SituationWithinRangeCM, and SituationAvailableCM are necessary. SituationWithinRangeCM gathers location information from GeoLocationCS in order to determine whether patients are nearby caregivers. SituationAvailableCM gathers status information from the CaregiveStatusCS in order to reason about the availability of a caregiver.

C. ECA-DL Rules

In order to delegate required pieces of reactive behaviour to the platform, we should specify ECA-DL rules that represent these behaviours. The following reactive behaviours are required:

1. *SeizureAlarm*: upon an epileptic seizure alarm, the service running on the patient's device should be notified of the possibility of an epileptic seizure and that the patient is currently performing a hazardous activity;
2. *HelpRequestNotification*: upon an epileptic seizure alarm, a collection of caregivers should be notified that a patient is in need of help. Only caregivers who are available and nearby the patient should receive the notification;
3. *HelpAcceptedNotificafion*: upon receiving an acceptance from a particular caregiver, the caregivers who had previously received the notification for help should be notified that one of the caregivers has already accepted to help that particular patient.

The following ECA-DL rule (ECARule1) aims at realizing the reactive behaviour *HelpRequestNotification*. This rule applies to all patients (scope clause). In addition, no condition (when clause) is defined in this rule. Upon receiving event notification of an epileptic alarm for any epileptic patient, action NotifyCaregiversServices should be invoked. The select clause selects all the patient's caregivers who are within range and available. The result collection of this select (collection of caregivers) is passed as one of the arguments of the action.

```
Scope (EpilepticPatient.*; p) {
   Upon EpilepticAlarm (p)
   Do NotifyCaregiversServices ( p, p.hasGeoLocation.coordinates,
          Select (Caregiver.*; care; isCaregiverOf (care, p) and
                 SituationWithinRange (p, care) and
                 SituationCaregiverAvailable (care))) }
```

The following ECA-DL rule (ECARule2) aims at realizing the reactive behaviour *HelpAcceptedNotification*. This rule uses the same scope clause as the previous rule. The upon clause specifies an event composition, in which Ev1 (EpilepticAlarm (p)) should occur followed by Ev2 (AcceptRequest (p)). This means that this composite event happens every time there is an epileptic seizure alarm, followed by an acceptance for helping the patient having the seizure. Upon the occurrence of this composite event, the action notifyAcceptanceCaregivers should be invoked. The select clause selects all the patient's caregivers who are within range, and are different from the caregiver who accepted the request. The identification of the caregiver who accepted the request is obtained by accessing one of the attributes of the event, namely Ev2.caregiverID. The result collection of this select is passed as one of the arguments of the action.

```
Scope (EpilepticPatient.*; p) {
   Upon Ev1: EpilepticAlarm (p); Ev2: AcceptHelpRequest (p)
   Do notifyAcceptanceCaregivers (
        Select (CareGiver.*; care; isCareGiverOf (care, p) and
                           SituationWithinRange (care, p) and
                           care <> Ev2.caregiverID), p, Ev2.caregiverID)}
```

IV. The Healthcare Service Prototype

Fig. 2 presents the prototype architecture. Components have been implemented using Java and Java RMI [6].

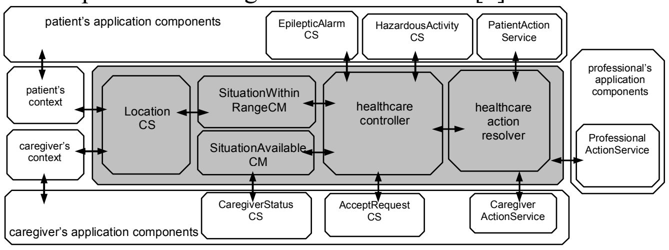

Figure 2 Service architecture

We have simulated the production of context information and events to evaluate the behaviour of the service under various load conditions. GeoLocationCS randomly generates geographical location coordinates for its users, respecting a maximum variation of the location changes. EpilepticAlarmCS and AcceptRequestCS also randomly generate EpilepticAlarm and AcceptRequest events, respectively. Similarly, the CaregiverStatusCS and HazardousActivityCS randomly changes the availability status of caregivers and whether the user is performing a potentially hazardous activity or not, respectively.

SituationCaregiverWithinRangeCM detects when patients and caregivers are nearby each other by means of Jess rules running on the local Jess engine. Similarly, the

SituationCaregiverAvailableCM component reasons about the availability of the caregivers by gathering context information from CaregiverStatusCS.

The healthcare controller component gathers context information from context sources and managers in order to obtain event notifications and context information values. We have implemented callback interfaces in the controller, which allows the context sources and managers to push context and situation information in the controller. When these callback methods are invoked, context and situation information is received by the controller and included into the local Jess working memory. ECA-DL rules run continuously on the local Jess engine.

Our prototype runs three instances of the Jess engine, one for each context manager component, and one for the controller component. Context sources are not implemented using Jess engines.

V. Performance Evaluation

In order to evaluate the scalability and the performance of the healthcare service that is built upon our context handling platform, we have defined the *reaction time* evaluation parameter. Reaction time is defined as the period of time between generating an event and finally invoking an action of an ECA-DL rule that refers to that event. The reaction time does not include the action execution time. For example, we can assess the reaction time between an EpilepticAlarm event occurrence and triggering ECARule1, or we can assess the reaction time between the occurrence of an AcceptRequest event and triggering ECARule2. Therefore, the reaction time consists of the processing time that takes place between an event occurrence, and the invocation of the action. We capture the reaction time in the EpilepticAlarmCS, which receives a callback invocation from the controller when rules ECARule1 and ECARule2 are triggered. These callback invocations are included in the RHS (Right Hand Side, which corresponds to the action part) of the Jess rules derived from ECARule1 and ECARule2, for the purpose of evaluating the performance. Therefore, every time one of these rules is executed, the Jess engine running in the controller component invokes the EpilepticAlarmCS component, which is then capable of calculating the reaction time. These callback invocations are not part of the normal service's behaviour.

A. General Configuration

In our evaluation we measured the reaction time between the occurrence of an EpilepticAlarm, and triggering ECARule1 and ECARule2. The reaction time between an occurrence of the EpilepticAlarm and triggering ECARule2 is longer than between the same EpilepticAlarm occurrence and triggering ECARule1. This is due to the composition of events defined in ECARule2, which requires that an event AcceptRequest occurs after the EpilepticAlarm.

We have collected reaction time measurements for different numbers of patients and caregivers (entities), and for different event generation rates (measured in events/second). For each pair (number of entities, number of events per second), we have collected 100 reaction time measurements, for which we calculate an *average reaction time* value. The average reaction time includes reaction times for ECARule1 and ECARule2.

We have performed stress tests in order to evaluate how the service behaves under extreme circumstances, such as with an extreme numbers of users and events. Stress testing aims at assessing the robustness and the availability of the system under heavy load. In order to perform stress testing, we have observed and measured the reaction times when the system is loaded with up to 35000 entities (patients and caregivers), and generates up to 450 events per second.

B. Centralized configuration

Fig. 3 shows the increase of the average reaction time in milliseconds (Y axis) with respect to the number of entities (X axis). Three curves are shown, each one representing a particular number of events generated per second. In these measurements all the components are executed in a single machine with a 1.86 GHz Pentium M processor and 1GB of RAM.

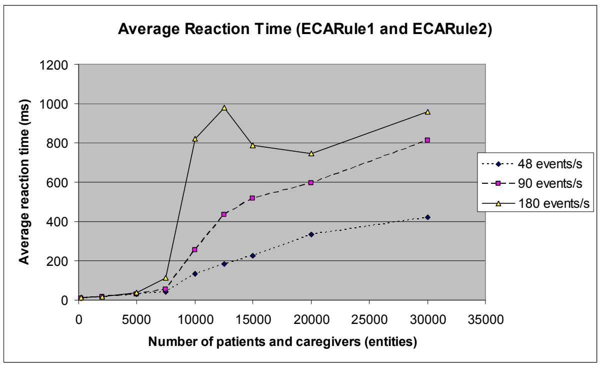

Figure 3 Average Reaction Time (ECARule1 and ECARule2) for the Centralized Configuration

We have asserted that about 85% of the reaction time consists of the processing time required by the Jess engine. The other 15% is due to the event consumption mechanism implemented by the controller component. The latter could be improved by optimizing the data structures to handle events, for example, by using hash tables.

When we increase the number of entities, the size of the working memory increases proportionally to the number of entities plus their context attributes, which should also be included in the working memory. The increase of the working memory (due to the representation of entities and their attributes) linearly degrades the performance of the Jess engine (as discussed in [5]), which consequently increases the reaction time. In addition, event notifications are also added to the working memory. Although they are not kept for long due to the event consumption and detection window interval, the addition and removal of events requires a rearrangement of the Rete network of nodes, which also consumes processing time, and increases the reaction time. These behaviours are reflected on the shapes of two curves in Fig. 3, namely the 45 events/s and the 90 events/s. For these curves, the reaction time is roughly linear with respect to the number of entities.

In the 180 events/s curve, for more than 7500 entities we see that the reaction time is highly unpredictable, which indicates a *saturation point*. This is because Jess cannot process the events fast enough to keep up with the frequency in which events are being generated (180 events/s). Therefore, event notifications may not be processed in the order in which they arrive, possibly leading to event starvation. To handle this saturation point, an event queuing mechanism may be necessary together with service performance management mechanisms. These mechanisms may either: (i) limit the load, e.g., by preventing the introduction of new users in the system; or (ii) cope with the load, e.g., by re-distributing controllers or adding processing resources.

C. Distributed Configuration

Fig. 4 depicts a graphic similar to the one in Fig. 3. For this experiment we distributed the components over two different machines: one with a 1.86 GHz Pentium M processor and 1GB of RAM; and the other with a 2GHz Core 2 Duo Mobile processor and 2GB of RAM, respectively. All context sources executed in the 1GB machine, while the context managers and the controller executed in the 2GB machine.

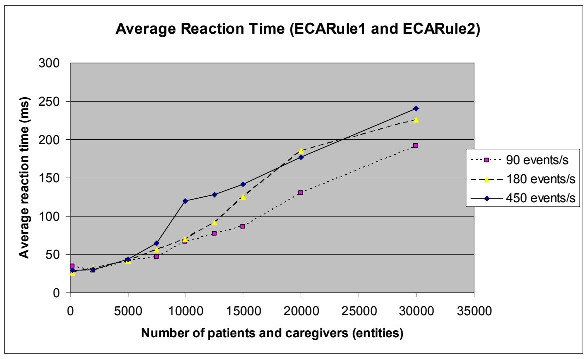

Figure 4 Average Reaction Time (ECARule1 and ECARule2) for the Distributed Configuration

Fig. 4 shows that the average reaction times in a distributed configuration improve considerably. For example, for 30000 entities and a rate of 90 events/s, the average reaction time is about 800 ms in the centralized configuration and 190 ms in the distributed configuration. In addition, the saturation point has not been reached even with a greater number of entities and a higher frequency of event generation. This behaviour can be explained by another characteristic of Jess, which is its memory usage. Jess is said to be a "memory-intensive application" [5], since Rete is an algorithm that explicitly *trades space for speed*. Therefore, the Jess' performance is strongly influenced by memory availability. In the distributed configuration, the three engines run on separate machines with increased memory availability, which improves the performance of these engines. In addition, the context sources consume a considerable amount of memory to simulate large amounts of context information and events. In the distributed configuration, the engines do not need to compete with the context sources for memory.

VI. Concluding Remarks

We have proposed an approach to the development of context-aware services by employing a rule-base realization. This approach is based on two related levels of design: (i) a specification level, which enables developers to define service behaviour at a high level of abstraction considering situations, events and their composition; and (ii) a realization level, which implements the service specification using a distributed rule-based approach for situation detection and rule execution.

In this paper we have demonstrated the suitability of our rule-based approach to support a highly dynamic context-aware service that includes reasoning for situation detection and event composition. We have developed a context-aware service in the domain of healthcare and evaluated its performance and scalability.

Performance is evaluated by collecting reaction time measurements from the healthcare prototype. Due to the Rete algorithm, Jess efficiently processes rules, which leads to satisfactory reaction times for our experiments (in the order of milliseconds). To the best of our knowledge, reaction times of this order would be acceptable for various types of context-aware services. The *scalability* of our approach has been assessed in the healthcare service by increasing the number of entities, the number of events generated per second, and the number of rules. We have also shown that the system behaves satisfactory for a large number of entities and events. In addition, the approach scales for an arbitrary number of ECA-DL rules, since ECA-DL rules can be distributed over several controller components, which can be added on demand.

References

[1] Freeband Kennisimpuls, "AWARENESS project". The Netherlands, 2004. Available at: http://awareness.freeband.nl

[2] Dockhorn Costa, Patrícia, Architectural Support for Context-Aware Applications: From Context Models to Services Platforms, CTIT Ph.D.-Thesis Series, No. 07-108, Telematica Instituut Fundamental Research Series, No. 021 (TI/FRS/021), The Netherlands, 2006.

[3] Dockhorn Costa, P. and Almeida, J.P.A. and Ferreira Pires, L. and van Sinderen, M.J. Situation specification and realization in rule-based context-aware applications. In: 7th IFIP WG6.1 Int'l Conf. Distributed Applications and Interoperable Systems, DAIS 2007, June 2007, Paphos, Cyprus. pp. 32-47. LNCS 4531. Springer.

[4] Dockhorn Costa, P. Ferreira Pires, L., van Sinderen, M.: Designing a Configurable Services Platform for Mobile Context-Aware Applications, International Journal of Pervasive Computing and Communications (JPCC), Vol. 1, No. 1. Troubador Publishing (2005)

[5] Friedman-Hill, E.: JESS in Action: Rule-Based Systems in Java. Manning Publications Co., (2003)

[6] Java Remote Method Invocation (Java RMI) website. Available at: http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

[7] OMG: Unified Modelling Language: Object Constraint Language version 2.0, ptc/03-10-04 (2003)

[8] OMG: UML 2.0 Superstructure, ptc/03-08-02 (2003)

[9] Roessingh Research and Development website. Available at: http://www.rrd.nl/www/indexa.html

[10] Sandia Labs, Jess: the Rule Engine for the Java Platform, Jess website. Available at: http://herzberg.ca.sandia.gov/jess/

[11] X. Hang Wang, D. Qing Zhang, T. Gu, H. Keng Pung, "Ontology-Based Context Modeling and Reasoning Using OWL", Proc. of the 2nd IEEE Conf. on Pervasive Computing and Communications Workshop (PERCOMW04), USA, 2004.