Situation Specification and Realization in Rule-Based Context-Aware Applications

Patrícia Dockhorn Costa[1], João Paulo A. Almeida[1,2],
Luís Ferreira Pires[1] and Marten van Sinderen[1]

[1]Centre for Telematics and Information Technology, University of Twente,
PO Box 217, 7500 AE Enschede, the Netherlands
[2]Computer Science Department, Federal University of Espírito Santo (UFES),
Av. Fernando Ferrari, s/n, Vitória, ES, Brazil
{dockhorn, almeida, pires, sinderen}@cs.utwente.nl

Abstract. Context-aware applications use and manipulate context information to detect high-level *situations*, which are used to adapt application behavior. This paper discusses the specification of situations in context-aware applications and introduces a rule-based approach to detect situations. Situations are specified using a combination of UML class diagrams and OCL constraints. We support a wide range of situations, which can be composed of more elementary kinds of context. We discuss how to cope with distribution and to exploit it beneficially for context manipulation and situation detection. We employ a generic rule-based platform (DJess [2]) to support the derivation of situations in a distributed fashion.

1 Introduction

Context-aware applications use and manipulate context information to detect the situations of users and adapt their behaviour accordingly. Context-awareness has become an important and desirable feature for ubiquitous computing, in which applications not only use context information to react on a user's request, but also take initiative as a result of (continuously-running) context reasoning activities. In this sense, ubiquitous context-aware applications can be characterized as *attentive* in addition to *reactive*. An example is an application that adapts the quality of audio and video streams automatically according to battery power consumption and the kind of network connectivity available, without user intervention.

The design of context-aware applications is a challenging task, which justifies the development of novel methods, abstractions and infrastructures (e.g., [1, 3, 4, 6, 14]). This paper proposes an approach to the specification and realization of situation detection for attentive context-aware applications. Our aim is to facilitate application design by providing abstractions for the specification of context-aware applications, in particular those related to the detection of situations based on context information. In order to detect situations attentively, a rule-based approach to situation detection is proposed. This solution is based on the use of a general-purpose rule-based platform, which guarantees the efficiency of situation detection (triggers upon situation detection as opposed to query-based solutions).

Situations are specified using standard UML 2.0 [18] class diagrams which are enriched with OCL 2.0 [17] constraints to define the conditions under which situations of a certain type exist. We support a wide range of situations, which can be composed of more elementary kinds of context. To transform the specification into a set of rules to be executed directly on the rule-based platform, we identify a number of patterns for rule detection realization. The rule set which is derived systematically from the specification can be deployed directly in the Jess rule engine. We employ the DJess distributed rule-based platform [2] to support the derivation of situations in a distributed fashion. This paper extends the work presented in [7], by discussing situation realization, detection and distribution.

The paper is further organised as follows. Section 2 discusses how context is modelled in our approach drawing on our previous work [7, 8]. Section 3 discusses the specification of situation types. Section 4 elaborates on the realization of situation detection with the help of rule engines. Section 5 discusses how situation detection can be done in a distributed fashion according to different scenarios. This is done to exploit distribution beneficially for context manipulation and situation detection, not only for scalability purposes but also to address information privacy concerns. Section 6 discusses related work, and, finally, Section 7 summarises our results and indicates future research.

2 Context Models

A *context-aware application* is a distributed application that adapts its behaviour according to its users' context. Figure 1 depicts a user interacting with a context-aware application. The application obtains context information from the user's environment (e.g., by means of sensor technology) in order to reason about context and detect situations of interest.

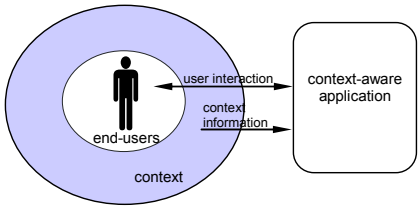

Figure 1. Users, their context and a context-aware application

Context can be defined as "the interrelated conditions in which something exists" [15]. This definition reveals that context is only meaningful with respect to a thing (that "exists"), which we call here an entity. The concept of entity is fundamentally different from the concept of context: context is what can be said about an entity, i.e., context does not exist by itself. Examples of entities are persons, computing devices and buildings. The context of an entity can have many constituents ("interrelated conditions"). Examples of some constituents of the context of a person are the person's location, mental state, and activity. In the remainder of this paper, we use the term context to refer to constituents of the context of an entity. Together, these constituents form the entity's context.

The process of identifying relevant context consists of determining the "conditions" of entities in the application's universe of discourse (e.g., a user or its environment) that are relevant for a context-aware application or a family of such applications. The representation of these relevant conditions or circumstances is called here a *context model*. We define a context model as a conceptual model (in the sense of [16]) of context. In previous work [7, 8], we have defined conceptual foundations that can be used beneficially in context modeling. These conceptual foundations include the separation of entity and context types, which are represented here as UML classes. We briefly discuss these foundations in the sequel. The work presented in [7, 8] provides a more detailed discussion.

Intrinsic Context

We characterize context as either *Intrinsic* or *Relational*. Intrinsic context defines a type of context that belongs to the essential nature of a single entity, i.e., it does not exist separate from this entity. In addition, intrinsic context does not depend on the relationship with other entities. Figure 2 depicts examples of intrinsic context types which could be used in many health-related applications. Geographic location (`GeoLocation`) is context that inheres in all spatial entities. Similarly, battery power (`BatteryPower`) inheres in a computing device (`Device`). Analogous reasoning can be applied to other context types depicted here.

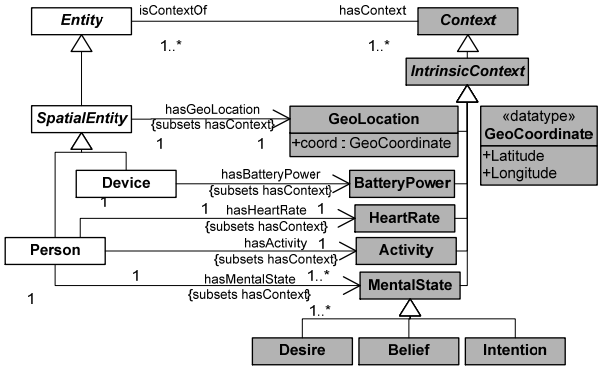

Figure 2. Intrinsic context types

Intrinsic context types are associated with a data type such that an instance of an intrinsic context type is assigned to a value of this data type. The geographical location of an entity is an example of intrinsic context type, whose data type consists of all possible values in a geographical coordinate system, represented by the `GeoCoordinate` datatype.

Relational Context

While intrinsic context inheres in a single entity, relational context inheres in a plurality of entities. Relational context may be used to relate an entity to the collection of entities that play a role in the entity's context. Figure 3 shows examples of

relational context types. The `NetworkAvailability` relational context type relates a device to a collection of networks that are available through that device, and `ChannelAvailability` relates a device to a collection of communication channels supported by that device (e.g., e-mail, voice and SMS).

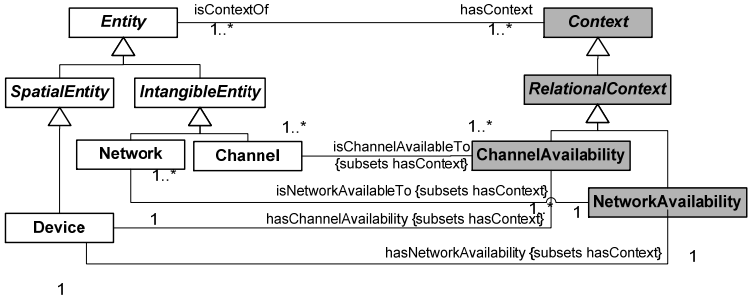

Figure 3. Relational context types

Some other examples of relational context are `DeviceAvailability` and `SocialNetwork`. The `DeviceAvailability` relational context relates a person to a collection of devices that are available to that person. `SocialNetwork` relates a person to the collection of persons interacting with that person by any communication channels.

3 Situation Models

The context models we have discussed so far, allow application designers to represent a context-aware application's universe of discourse. This section introduces *situation models*, which explicitly represent particular situations of interest, given a certain context model.

Situations define particular states of affairs which are of interest to applications. They are composite concepts whose constituents are the elements of our context models, i.e., entities, and intrinsic and relational contexts. In this sense, situation models should extend and comply with the context models. For example, a situation model can represent the situation in which "John is near Alice and their mobile phones are available" or "John has a fever and influenza". The underlying context model for this example should define that a person may be near another person and that a person may own a mobile phone.

In our approach we define *situation types*, which aim at characterizing situations with similar properties. For example, the situation type "John is within 50 meters from Alice" consists of all situation instances in which the distance between John's and Alice's location values is less than 50 meters. Similarly, the situation type "Person is within 50 meters from another person" consists of all situation instances in which the distance between any two persons' location values is less than 50 meters. Although unanticipated situation instances are supported at application runtime, situation types are defined at application design-time.

The examples used throughout the paper illustrate a range of situation patterns that are relevant for context-aware applications. These patterns involve the different kinds

of context (intrinsic and relational) and entities, which are the building blocks used to compose situations. We use a combination of UML class diagrams and OCL constraints to specify situations.

3.1 Situations involving intrinsic context

Situations involving intrinsic context are composed by a unique entity and part of its intrinsic context. The following example represents a situation type (`SituationAvailable`) that captures the availability and willingness to communicate of MSN and Skype users. Figure 4 depicts a fragment of the structural context model that represents the `MsnStatus` and `SkypeStatus` intrinsic context types, which model the user's communication status while using MSN and Skype, respectively. A person, while playing the role of `MsnUser`, is associated with `MsnStatus` context type, and while playing the role of `SkypeUser`, is associated with `SkypeStatus` context type. The enumeration data types `SkypeStatusEnum` and `MsnStatusEnum` define all possible values for `SkypeStatus` and `MsnStatus`, respectively.

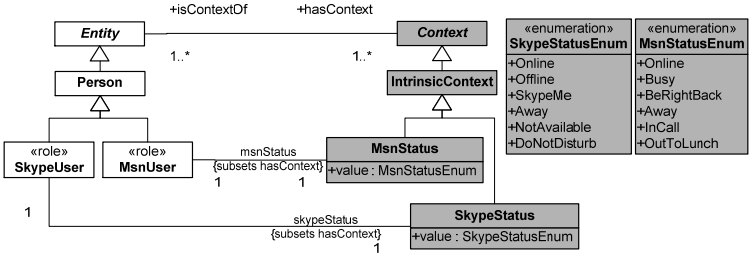

Figure 4. Fragment of context model

Figure 5 depicts a situation model which builds on the context model presented in Figure 4, defining the situation type `SituationAvailable`.

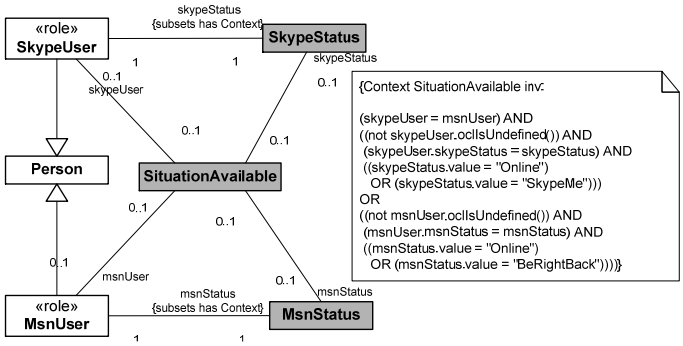

Figure 5. SituationAvailable specification

The OCL invariant in this diagram is a predicate that must hold for all instances of `SituationAvailable`. It defines that instances of `SituationAvailable` must be either

associated with a user available in Skype (with `SkypeStatus` set to `Online` or `SkypeMe`) or a user available in MSN (with `MsnStatus` set to `Online` or `BeRightBack`). The OCL operation `oclIsUndefined()` is part of the OCL standard library and tests whether the value of an expression is undefined.

Figure 6 shows an example of situation involving two entities and their intrinsic context. Their locations are compared such that instances of `SituationWithinRange` hold if two persons are located within a certain range (defined as an attribute of the `SituationWithinRange` class). This model builds on the context model defined in Figure 2.

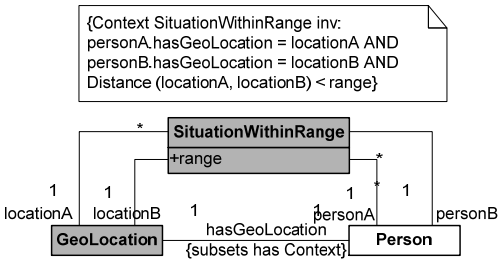

Figure 6. SituationWithinRange specification

3.2 Situations involving relational context

Situations involving relational context consist of at least two entities and part of their relational contexts. The following example discusses a situation in which a device has established a connection (relational context type) to each of the two network types, `WLAN`, and `Bluetooth` (entities). By explicitly modeling the connections as relational context, we are able to assign properties to these connections, such as access rights and negotiated QoS.

Figure 7 depicts the structural context models representing the types and relationships that are relevant for this example. According to this diagram, a `Device` may be connected to a `Network` through the relational context `Connection`.

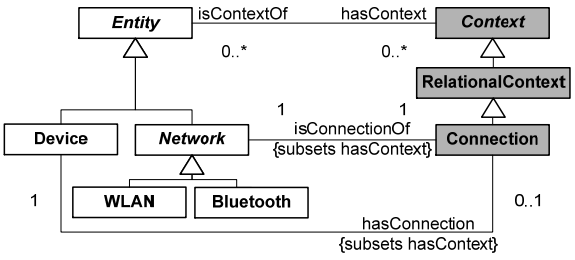

Figure 7. Fragment of context model

Figure 8 depicts the situation type `SituationConnected`. The OCL invariant defines that instances of this situation must be associated with at least one connection object.

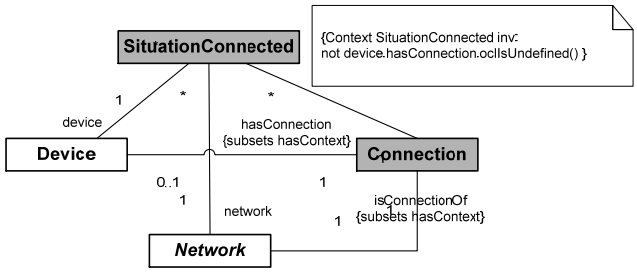

Figure 8. SituationConnected specification

3.3 Situation of situations

Situation themselves may be composed of other situations. Suppose we would like to know when a device switches from a `WLAN` connection to a `Bluetooth` connection in order to set new quality of service parameters. Since `SituationConnected` has been already defined in Figure 8, in order to detect `SituationSwitch`, we would have to verify whether `SituationConnected` held in the past for network `WLAN`, and currently holds for network `Bluetooth`. We may add the additional constraint that the handover time should not be longer than one second. This example is depicted in Figure 9, showing that `SituationSwitch` can be modeled by composing multiple occurrences of `SituationConnection`, one called `wlan`, and the other called `bluetooth`.

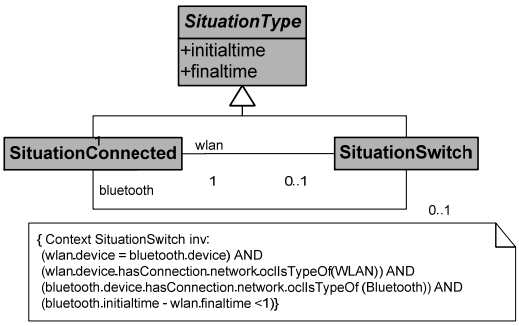

Figure 9. SituationSwitch specification

This situation requires using temporal aspects, which are represented in our approach by means of initial and final times. Each situation type extends the `SituationType` class inheriting these temporal attributes. The `initialtime` attribute captures the moment a situation begins to hold, and the `finaltime` attribute, the moment a situations seizes to hold. Since we capture the `finaltime`, our model represents past

occurrences of situations[1]. We also include temporal operations for relating situations in their occurrence intervals, such as precedence, overlapping, and post-occurrence. These operations are defined in OCL in terms of initial and final times, and can be used in the definition of situations.

4 Rule-Based Implementation

In a rule-based implementation, the designer defines rules which are applied to facts in a working memory. The mechanism used for rule application (and in our case situation detection) is based on the Rete algorithm [11], which efficiently matches the patterns for situations by remembering past pattern matching tests. Only new or modified facts are tested against the rules.

Figure 10 depicts the elements of our approach with the correspondence between the UML specification, the Java code and the Jess code at the template level (design-time). At the instance level (runtime), Figure 10 depicts the relations between the user's context and the rule-based implementation. Context sources provide context information, which is input as facts in the engine's working memory.

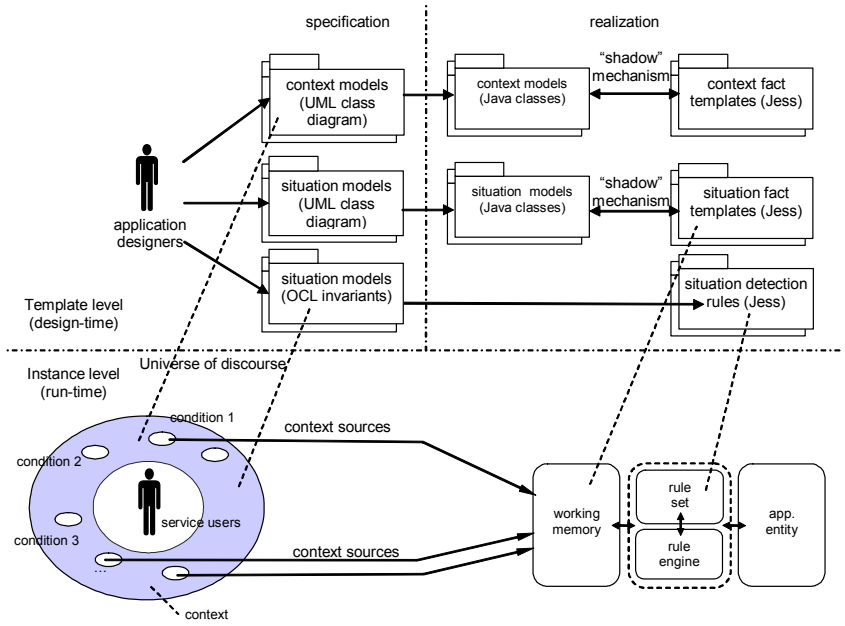

Figure 10. Correspondences between UML specifications, and Java and Jess code

[1] The invariants as presented in the figures are violated for past occurrences of situations. In order to avoid that, we should include, for each invariant, a disjunction with a predicate that verifies whether this situation is a past occurrence (not finaltime.oclIsUndefined()). We omit this predicate in this paper for the sake of readability.

We have used *shadow facts* to implement our structural context models. This is a mechanism offered by Jess to serve as a connection between the working memory and a Java application. Objects created in Java are reflected in the working memory. Therefore, any alteration of the Java objects is automatically perceived by the Jess working memory. The Java classes in our implementation directly reflect the UML models defined at the context model, such that their generation can be automated. We have used Octopus (http://www.klasse.nl/octopus) for generating Java code from UML2.0 class diagrams.

Once we have defined the structural context models, we can carry out the situation detection realization. Similarly to the structural context model, each situation type, as specified in the UML class diagram, corresponds to a Java class, as well as a shadow fact template. Situation instances are represented as shadow facts that are created and deactivated by rules for situation detection. Each situation type leads to the definition of two rules, namely a rule for situation fact creation, and a rule for situation fact deactivation. Conditions for enabling these rules are derived from the invariants of situation classes. The rule for situation creation detects when an invariant becomes true, and the rule for situation deactivation detects when the invariant becomes false. We have identified patterns of situation types that are systematically mapped to Jess code. Automatic code generation from OCL to Jess is work in progress.

A situation fact life cycle consists of creation, activation, deactivation and destruction. The activation of a situation fact occurs simultaneously to its creation, and the deactivation occurs when the situation invariant no longer holds. Figure 11 uses a UML 2.0 activity diagram to show when situations should be created or deactivated. When the invariant holds and the situation fact does not exist yet, the situation fact is created; when the invariant no longer holds, the situation fact is deactivated.

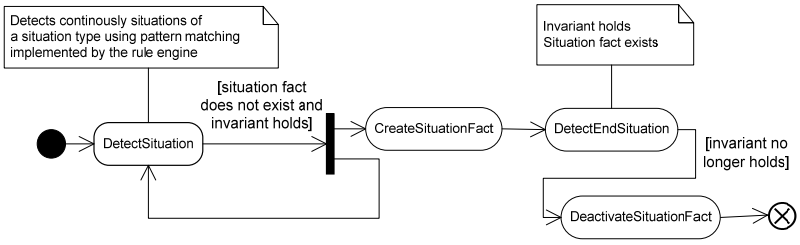

Figure 11. Activity diagram for situation creation and deactivation

Deactivated situation facts consist of historical records of situation occurrence, which may be used to detect situations that refer to past occurrences. Currently, we implement a simple rule-based time-to-live mechanism for historical records, which considers the final time of deactivated situation facts. We have identified that situation realization in Jess follows certain patterns of implementation. Table 1 depicts how creation and deactivation rules should be formulated.

These rules are written in the Jess language. Conditions and actions are separated by the symbol "=>". The condition part (or left hand side) consists of patterns that match facts in the working memory. A pattern is represented in between parentheses,

such as `(situation type invariant)`. The action part of a rule (or right hand side) contains function calls, such as the functions to create and to deactivate situations.

Table 1. Creation and deactivation rules

Creation Rule	Deactivation Rule
`(situation type invariant)`	`(not (situation type invariant))`
`(not (situation exists))`	`(situation exists)`
`=>`	`=>`
`create (situation)`	`deactivate (situation)`
`[RaiseEvent()]`	`[RaiseEvent()]`

The condition part of a creation rule checks whether the OCL invariant holds, and whether there is already an instance of that particular situation currently active (final time not nil). If these conditions are met, a situation fact is created, and optionally, an event can be raised. Analogously, the condition part of a deactivation rule checks whether the OCL invariant no longer holds, and there is a current situation fact active. When these conditions are met, this situation instance is deactivated, and optionally, an event can be raised. Figure 12 depicts how `SituationConnected` and `Situation-Switch` (see section 3) are implemented in Jess.

```
;Creation rule (SituationConnected)
(defrule create_situation_connected
    (Device (OBJECT ?dv) (hasContext ?contexts) (sizeContexts ?s))
    (test (?dv hasContextType "context_control.Connection"))
    (not (SituationConnected (OBJECT ?st) (device ?dv) (finaltime nil)))
   =>
    (bind ?SituationConnected (new situation_control.SituationConnected ?dv))
    (definstance SituationConnected ?SituationConnected))

;Deactivation rule (SituationConnected)
(defrule deactivate_situation_connected
    (Device (OBJECT ?dv) (identity ?id) (hasContext ?ctxs) (sizeContexts ?size))
    (test (not (?dv hasContextType "context_control.Connection")))
    (SituationConnected (OBJECT ?st) (device ?dv) (finaltime nil))
    =>
    (call ?st deactivate))

;Creation rule (SituationSwitch)
(defrule create_situation_switch
    (Device (OBJECT ?dv) (identity ?dvid))
    (SituationConnected (OBJECT ?SWlan)
                        (device ?device&:(eq (call ?device getIdentity) ?dvid))
                        (network ?net&:(instanceof ?net context_control.WLAN))
                        (finaltime ?finaltime&:(neq ?finaltime nil)))
    (SituationConnected (OBJECT ?SBlue) (device ?dv)
                        (network ?net2&:(instanceof ?net2 context_control.Bluetooth))
                        (starttime ?start) (finaltime nil))
    (test (<= (- (call ?start getTime)(call ?finaltime getTime)) 60000))
    (not (SituationSwitch (OBJECT ?st) (wlan ?SWLAN) (bluetooth ?SBlue)
                          (finaltime nil)))
    =>
    (bind ?SituationSwitch (new situation_control.SituationSwitch ?SWlan ?SBlue))
    (definstance SituationSwitch ?SituationSwitch))
```

Figure 12. Situation realization in Jess

The condition part of the `create_situation_connected` rule checks whether there is a `Connection` relational context in the list of contexts of that device. This part of the

condition corresponds to the OCL invariant defined in Figure 8. In addition, it checks whether a `SituationConnected` instance does not already exist for that device. When these conditions are met, the action part is triggered, i.e., an instance of `SituationConnected` is created for that device.

The condition part of the `deactivate_situation_connected` rule, on the contrary, checks whether the device is no longer connected to a network, and if there is an existing `SituationConnected` for that device. If these conditions are met, that particular instance of `SituationConnected` is deactivated, and can be used in the future as a historical record.

The condition part of the `create_situation_switch` rule checks whether there was an instance of `SituationConnected` with network `WLAN` in the past (`finaltime` not nil), and currently there is an instance of `SituationConnected` with network `Bluetooth`. In addition, the handover time should not be longer than 60 seconds. These parts of the condition correspond to the OCL invariant depicted in Figure 9. As in all creation rules, the condition also checks whether there is no instance of `SituationSwitch` for that particular handover currently active. When these conditions are met, an instance of `SituationSwitch` is created. We did not include here the `deactivate_situation_switch` rule due to the lack of space.

To allow maintenance of past situations, we use a mechanism based on object serialization to preserve the situation state at the time the situation was deactivated. When a situation is deactivated, a serialized copy of the situation is created and stored for future use. Serialized objects are given unique identifiers, so that they can be retrieved unambiguously. For this reason, when checking the existence of a past instance of `SituationConnected`, we have used an unique identifier of the device `(call ?device getIdentity)`, instead of the object identifier `(device ?dv)` as in the currently active instance.

5 Distribution Issues

So far, we have focussed on the various rule patterns for the detection of the various kinds of situation. We have presented the realization solutions without regard for distribution, as if situation detection were based on a single rule engine, working with a single set of rules and a single working memory. In this section, we consider alternative distribution scenarios, and discuss their trade-offs.

Firstly, we consider the fully centralized scenario, in which no distribution is employed. In this scenario, context sources feed context information into the central rule engine's working memory, as depicted in Figure 13. This is the simplest scenario, and has limited scalability with respect to the number of situations detected, even when situations are entirely independent of each other, i.e., when situations are detected using context conditions that are sensed independently, and are not composed of other situations. The centralized approach introduces a single point of access to context information, which can be considered a potential (privacy) hazard, due to the sensitive nature of particular kinds of context information.

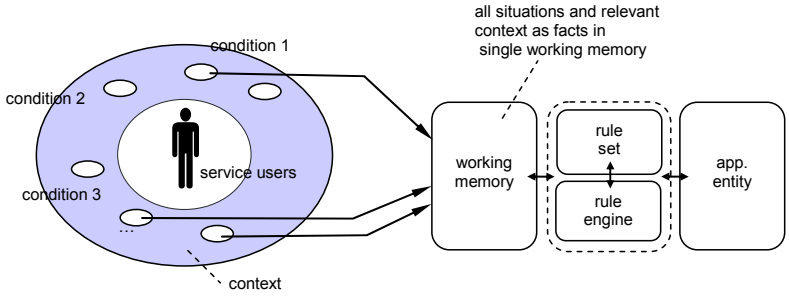

Figure 13. Centralized scenario

Secondly, we consider a scenario with multiple hub-and-spokes for situation detection. In this scenario, multiple engines detect independent situations. The level of distribution is constrained by the nature of the situation model, each hub-and-spoke pattern consisting of a centralized solution. In this approach, each rule engine may be associated to a different administrative domain, which enables more fine-grained control of the (privacy) policies which apply to the context information for that domain. The solution is highly constrained by the nature of the situation model, since all related situations must be detected in the scope of the same rule engine. Figure 14 depicts this solution with two rule engines detecting independent situations 1 and 2.

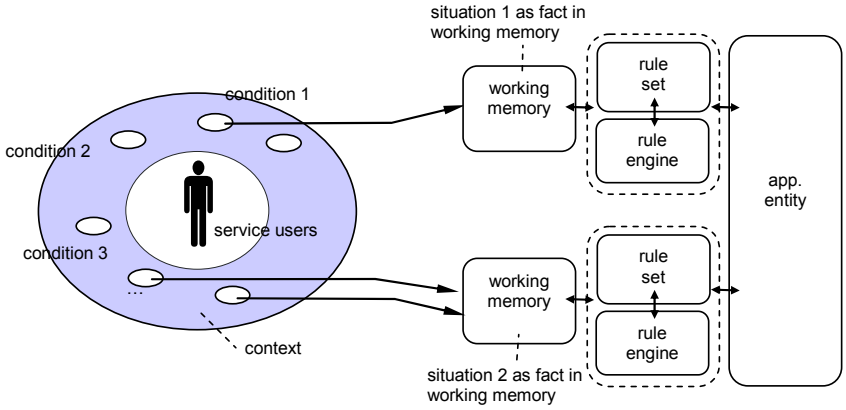

Figure 14. Multiple hub-and-spokes scenario

Finally, we consider a distribution scenario with a higher level of distribution that not only exploits possible independent situations, but that is able to decompose situation detection further, and distribute parts of the rule detection functionality to different rule engines. Different distribution strategies and rule engine configurations can be accommodated using this approach. Figure 15 depicts a possible configuration with two independent situations 1 and 2 detected independently in rule engines A and B (as in the hub-and-spokes scenario). The facts corresponding to those situations are shared with a rule engine C, which detects a situation 3 which is derived from situa-

tions 1 and 2. We propose a shared working memory mechanism that is part of the DJess infrastructure [2] to realize this approach. With this mechanism, rule engines running in different nodes can apply rules on shared sets of facts. A rule engine may participate in multiple shared memory partnerships (which are called Web of Inference Systems in DJess), each of which defining a shared set of facts, thus allowing arbitrary configurations.

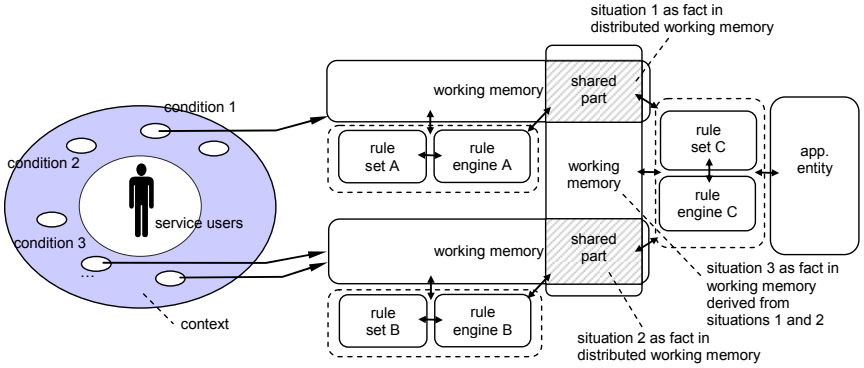

Figure 15. Distributed scenario

The distributed scenario enables fine-grained control of the policies that apply to context information, since different rule engines and parts of situation detection can be associated with different administrative domains. The policies for context information may justify in this scenario different distribution strategies. For example, consider an application that uses the distance between two users to determine whether users can view each other's contact information. Suppose further that GPS location is used to compute the distance between users. Due to the sensitive nature of the "raw" GPS location, different policies apply to this information, and to the aggregate and usually less sensitive distance information. In this case, GPS location should be only available to the engines that derive proximity information. Only the aggregated proximity information should be shared with other engines that define contact information visibility.

6 Related Work

Several approaches presented in the literature [12, 13, 19] support the concept of situation as a means of defining particular application's states of affairs. These approaches usually apply centralized mechanisms, and instead of detecting situations attentively, they offer reactive query interfaces, which do not take the initiative of generating events upon situation detection.

The work presented in [13] discusses a situation-based theory for context-awareness that allows situations to be defined in terms of basic fact types. Fact types are defined in an ORM (Object-Role Modeling) context model, and situation types are defined using a variant of predicate logic. The realization supported by means of a mapping to relational databases, and a set of programming models based on the Java

language. Although CML is based on a graphical notation, to the best of our knowledge, there is no modeling tool available for graphical situation specification. In addition, the implementation, as reported in [13], does not consider situation detection distribution.

None of the approaches we have studied use UML 2.0 in combination with OCL invariants for defining situation types. UML is unfairly underestimated in the context-awareness community. As we have seen in this paper, UML can be an appropriate and effective tool for modeling context and situations types. Furthermore, UML is currently widely adopted as a general modeling language, with extensive documentation and tool support.

7 Conclusions

We have proposed a novel approach for the specification and realization of situation detection for attentive context-aware applications. The specification approach is based on our earlier work on conceptual modeling for context information, and uses standard UML class diagrams for graphical representation of context models and situation models. Situations can be composed of more elementary kinds of context, and in addition can be composed of existing situations themselves. We have addressed the temporal aspects of applications, and included primitives to relate situations based on their temporal aspects.

The realization is rule-based, and executes on mature and efficient rule engine technology available off-the-shelf. The rule set is derived systematically from the specification and has been deployed directly in the Jess rule engine. We have argued that a distributed solution to situation detection has benefits, which apply to context-aware applications in particular. We have realized communication between rule engines by using the DJess shared memory mechanism, which allows different engines to execute their rule base in a shared set of facts.

This work is part of a larger effort towards a generic infrastructure to support context-aware applications. The use of a rule-based approach enables us to perform situation detection efficiently, and to generate events for situation detection with little effort. In addition, we also apply rule-based approaches to implement Event-Condition-Action (ECA) rules in our infrastructure [9].

As part of future work, we intend to study more complex mechanisms for discarding historical situation records that will no longer be used. Our current solution uses time-to-live for discarding historical records. An alternative solution is to eliminate all historical data that is not referred by any active situation. This requires complex inspection on situation type dependencies.

Acknowledgements

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.

References

1. Almeida, J.P.A., Iacob, M.E., Jonkers, H., and Quartel, D.: Model-Driven Development of Context-Aware Services. In: Distributed Applications and Interoperable Systems (DAIS 2006), 6th IFIP International Conference, LNCS, vol. 4025, Springer (2006) 213–227
2. Cabitza, F., Sarini, M., Dal Seno, B.: DJess - a context-sharing middleware to deploy distributed inference systems in pervasive computing domains. In: Proceeding of International Conference on Pervasive Services (ICPS '05), IEEE CS Press (2005) 229–238
3. Dey, A. K., Salber, D., and Abowd, G. D.: A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. Human-Computer Interaction, 16(2-4) (2001) 97–166
4. Chen, H. Finin, T., Joshi, A.: An ontology for context-aware pervasive computing environments, Knowledge Engineering Review, Special Issue on Ontologies for Distributed Systems, Vol. 18, No. 3. Cambridge University Press (2003) 197–207
5. Dockhorn Costa, P. Ferreira Pires, L., van Sinderen, M.: Architectural Support for Mobile Context-Aware Applications. In Handbook of Research on Mobile Multimedia, Idea Group Inc. (2005)
6. 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)
7. Dockhorn Costa, P., Guizzardi, G., Almeida, J.P.A., Ferreira Pires, L., van Sinderen, M.: Situations in Conceptual Modeling of Context. In Workshop on Vocabularies, Ontologies, and Rules for the Enterprise (VORTE 2006) at IEEE EDOC 2006, IEEE CS Press (2006)
8. Dockhorn Costa, P., Almeida, J.P.A., Ferreira Pires, L., Guizzardi, G., van Sinderen, M.: Towards Conceptual Foundations for Context-Aware Applications. In: Proc. of the Third Int'l Workshop on Modeling and Retrieval of Context (MRC'06), Boston, USA (2006)
9. Etter, R., Dockhorn Costa, P., Broens, T.: A Rule-Based Approach Towards Context-Aware User Notification Services. Proc. of the IEEE International Conference on Pervasive Services 2006, Lyon, France (2006)
10. Freeband A-MUSE Project, http://www.freeband.nl/project.cfm?id=489
11. Friedman-Hill, E.: JESS in Action: Rule-Based Systems in Java. Manning Publications Co., (2003)
12. Hang Wang, X., Qing Zhang, D., Gu, T., Keng Pung, H.: Ontology-Based Context Modeling and Reasoning Using OWL. Proc. of the 2nd IEEE Annual Conf. on Pervasive Computing and Communications Workshops (PERCOMW04), USA (2004) 18–22
13. Henricksen, K., Indulska, J.: Developing context-aware pervasive computing applications: Models and approach. Journal of Pervasive and Mobile Computing, volume 2(1), pages 37-64. Elsevier (2006)
14. McFadden, T., Henricksen, K., Indulska, J., Mascaro, P.: Applying a Disciplined Approach to the Development of a Context-Aware Communication Application. In: 3rd IEEE Conf. on Pervasive Computing and Communications (Percom 2005), IEEE CS Press (2005)
15. Merriam-Webster, Inc.: Merriam-Webster Online; http://m-w.com.
16. Mylopoulos, J.: Conceptual modeling and Telos. In: P. Loucopoulos and R. Zicari, eds, Conceptual modeling, databases, and CASE, John Wiley and Sons Inc., New York (1992)
17. Object Management Group: Unified Modelling Language: Object Constraint Language version 2.0, ptc/03-10-04 (2003)
18. Object Management Group: UML 2.0 Superstructure, ptc/03-08-02 (2003)
19. Strang, T., Linnhoff-Popien, C., and Frank, K.: CoOL: A Context Ontology Language to enable Contextual Interoperability. In: Proc. of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS2003), Paris (2003) 236–247