A Model-Driven Approach to Situations:

Situation Modeling and Rule-Based Situation Detection

Patrícia Dockhorn Costa, Izon Thomas Mielke, Isaac Pereira, João Paulo A. Almeida
Computer Science Department, Federal University of Espírito Santo (UFES)
Av. Fernando Ferrari, s/n, Vitória, ES, Brazil
pdcosta@inf.ufes.br, izontm@gmail.com, pereira.zc@gmail.com, jpalmeida@ieee.org

Abstract— **This paper presents a model-driven approach to the specification of situations and situation detection. We offer two main contributions: (i) a Situation Modeling Language (SML), which is a graphical language for situation modeling, and (ii) an approach to situation detection based on the transformation of a SML model into a set of rules to be executed on a rule-based platform. We exemplify our situation-based development approach with an application scenario in the domain of (mobile) banking, in which situations for detecting fraud-susceptible behavior are defined in SML. Based on the SML models, we discuss the rules that can be deployed on Drools for situation detection. The approach supports situation types defined in terms of patterns of facts, as well as complex situations in terms of reusable situation types, both at the specification level and realization level.**

Keywords: situation specification; situation detection; rule-based implementation

I. INTRODUCTION

The ability of a system to perceive and react to situations of interest can be broadly referred to as *situation awareness*. Situation awareness requires the ability to perceive facts and to identify in these the patterns that characterize situations of interest. Situation awareness has received increasing attention in the latest years, e.g., in the areas of *human-computer interaction* [10], in which systems are designed to facilitate human situation awareness; in *context-aware ubiquitous systems* [5], [6], in which situations characterize the user's context to promote application adaptation; in *autonomic computing*, in which a system's "self-situation" [9] is monitored to support the planning of "self-managing" actions; and in *context-aware business process modeling* [22], in which the "combination of all situational circumstances that impact process design and execution can be termed the *context* (aka situation) in which a business process is embedded." [21].

As discussed by Kokar et al. in [15], "to make use of situation awareness […] one must be able to recognize situations, […] associate various properties with particular situations, and communicate descriptions of situations to others." The notion of situation enables designers, maintainers and users to abstract from the lower-level entities and properties that stand in a particular situation and to focus on the higher-level patterns that emerge from lower-level entities in time.

We argue that enterprise systems can profit from the notion of situation and its adequate support both at design-time and at run-time. At *design-time*, behavior and policies can be defined in terms of the types of situations in which they apply, instead of various low-level conditions. This not only fosters separation of concerns through abstraction but also enables the definition of complex situation types by reusing previously defined situation types. At *run-time*, sophisticated situation detection machinery can be employed, enabling timely reaction to current situations.

In order to leverage the benefits of the notion of situation, proper support is required at the modeling level and at the realization level. We address this with a model-driven approach to the specification of situations and realization of situation detection. We offer two main contributions: (i) a Situation Modeling Language (SML), which is a graphical language for situation modeling, allowing the expression of primitive situations and complex situations involving the composition of situations (with temporal constraints when required), and (ii) an approach to situation detection based on the transformation of a SML model into a set of rules to be executed on a rule-based platform.

We exemplify our situation-based development approach with an application scenario in the domain of (mobile) banking, in which situations for detecting fraud-susceptible behavior are defined in SML. Based on the SML models, we discuss the rules that can be deployed on Drools for situation detection.

The paper is further structured as follows: Section II briefly characterizes situations and sets requirements for our approach; Section III presents a scenario which is used to exemplify the application of the approach; Section IV introduces the Situation Modeling Language (SML); Section V presents a formal semantics for SML; Section VI discusses rule-based situation detection; Section VII discusses related work and, finally, Section VIII presents concluding remarks and indicates points for future investigation.

II. SITUATIONS

Situations are composite entities whose constituents are other entities, their properties and the relations in which they are involved [6]. Situations support us in conceptualizing certain "parts of reality that can be comprehended as a whole" [19]. Examples of situations include "John is working", "John has fever", "John is working and has fever", "John and Paul are outdoors, at a distance of less than 10m from each other", "Bank account number 87346-0 is overdrawn while a suspicious transaction is ongoing", etc. (Technically, the sentences we use to exemplify situations are utterances of propositions which hold in the situations we consider; however, we avoid this distinction in the text for the sake of brevity.)

Situations are often reified (such as in [3], [6]), or ascribed an "object" status [15], which enables one not only to identify situations in facts but also to refer to the properties of situations themselves. For example, we could refer to the duration of a particular situation or whether a situation is current or past, which would enable us to say that the situation "John has fever" occurred yesterday and lasted two hours. The temporal aspect of situations also enables us to refer to change in time, thus we could say that "John's temperature is rising" or that "Account number 87346-0 has been overdrawn for the last 15 days".

A situation type [15] (or what is called *situoid* universal in [18], [19]) enables us to consider general characteristics of situations of a particular kind, capturing the criteria of unity and identity of situations of that kind. An example of situation type is "Patient has fever". This type is multiply instantiated in the cases in which instances of "Patient" (such as "John", "Paul", etc.) can be said to "have fever". Thus "John has fever" and "Paul has fever" are examples of instances of "Patient has fever". These examples reveal the need to refer to entity types such as "Patient" as part of the description of a situation type. The same can be said for "has fever" which, in this case, is defined in terms of a property of entities which instantiate the entity type "Patient" (namely "body temperature"). Detecting situations (i.e., instantiations of a situation type) require detecting instances of the entity types involved in the situation whose properties satisfy constraints captured in the situation type.

These characteristics of situations lead us to the following basic requirements for our situation-based approach:

1. Situation types should be defined at design time, and situations instantiating these types should be detected at runtime;
2. Situation types should be defined with reference to entity types as well as constraints on entities' properties and relations;
3. Temporal properties of situations should be considered (such as initial time, and, for a past situation, final time and duration).

In addition to these requirements, we have also observed that the definition of complex situation types may be more manageable by defining these types in terms of a composition of simpler situation types. Thus, we also include the recursive composition of situation types in our approach.

III. APPLICATION SCENARIO

We consider a banking application scenario in which the user can access his/her account via mobile devices, personal computers and ATMs. We intend to detect fraud-susceptible behavior, such as the ones generated from password theft and bankcard cloning.

Typically, fraud-susceptible behavior detection lies on tracking of the user's profile, which is considered together with incoming transactions, looking for unusual situation patterns. Some examples of well-established patterns to detect suspicious behavior are: (i) unusual credit-card debits on unlikely product groups; (ii) large transfer transactions to unrelated accounts; (iii) the amount of withdrawals over a day is N-times larger than the average amount of withdrawals over a month, etc. In our scenario, we consider context information (e.g. time and location), in order to detect behavior patterns that can be characterized as suspicious. An example of a suspicious situation is when an account is accessed from two different locations (e.g., 300 kilometers apart from each other), within a short period of time (e.g., 10 minutes). When this situation occurs, appropriate action can be performed, such as, e.g., notification of the account owner and auditing of transactions in this account.

Before we proceed to discuss situation modeling, we should first characterize the basic elements in the domain of discourse of our scenario. For that, we employ a so-called context conceptual model [6] which defines the basic vocabulary to be used in the subsequent definition of situations (see Figure 1). We use a profiled UML class diagram incorporating in UML the basic distinctions proposed in [7]. The stereotype «Entity» is used to model entities which are considered to exist independently of other entities. The entity types considered in this scenario are *Device* and *Account*. The stereotype «IntrinsicContext» is used to model a property of an entity (which is existentially dependent on a single entity). The context type *Location* is an example of intrinsic context that is used to characterize a *SpatialEntity* (*Device*). The stereotype «RelationalContext» is used to model relations which are established between entities (and depend existentially on the entities related). The relational context *Access* relates *Device* and *Account*, capturing the relationship established when an *Account* is being accessed through a *Device*. The relational context *OngoingWithdrawal* captures the relationship that exists when the user of an *ATM* requests a withdrawal of a certain *Account* until the end of the transaction (when cash is dispensed or the transaction is cancelled).

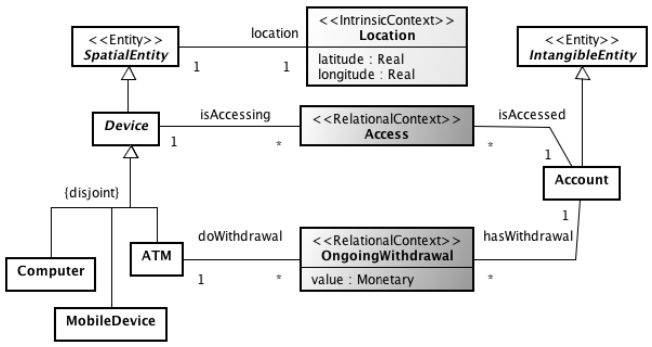

Figure 1 Context model for the banking scenario

The most basic situation type we are interested in is that in which an account is being accessed (*LoggedIn*). Further, we are interested in the situations in which a suspicious withdrawal is ongoing; a suspicious withdrawal is defined as a withdrawal of value greater than $1000,00 (*OngoingSuspiciousWithdrawal*). We are also interested in two situation types that involve temporal relations between instances of *LoggedIn*: *SuspiciousParallelLogin, SuspiciousFarAway-Login* and *AccountUnderObservation*. We will address the specification of these situations in Section IV.

IV. THE SITUATION MODELING LANGUAGE (SML)

In this section we discuss the concepts used in our proposed modeling language by means of the SML models of situation types previously mentioned.

Figure 2 depicts the definition of the *LoggedIn* situation type, graphically represented as a rounded rectangle. The elements composing a particular situation are shown inside the bordered rectangle that represents that situation. In this case they are the entity types *Device* and *Account*, and the relational context type *Access*. Entity types are represented as rectangles and Relational context types as diamonds. Relational Context types and Entity types are connected by a directed arrow, which has a role name consistent with names specified in the context model, as depicted in Figure 1. Finally, the small diamonds at the border represent the entities of this situation which may be referred to by more composite situations.

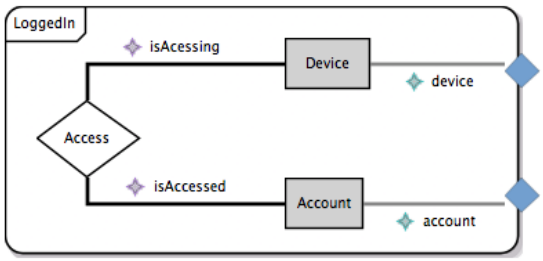

Figure 2. Situation LoggedIn

Figure 3 defines the *OngoingSuspiciousWithdrawal* situation type, which captures the situation in which an ATM performs a suspicious withdrawal from an account. The entities participating in this situation type are *ATM* and *Account*, which are connected by a relational context type *OngoingWithdrawal*. A suspicious withdrawal is defined by constraining the *value* attribute of the *OngoingWithdrawal* relational context to be greater than $1000,00, which is a literal.

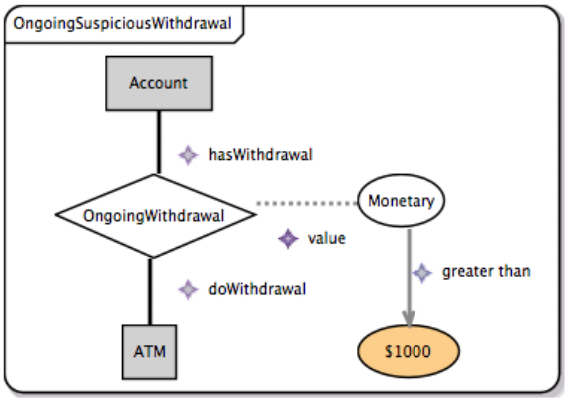

Figure 3. Situation OngoingSuspiciousWithdrawal

Attribute types, such as *Monetary*, are represented with white oval shapes, while literals are represented with orange oval shapes. The *greater than* directed arrow between the *Monetary* value attribute type and the literal is an application of the built-in *greater than* relation, which is a *formal relation*, in the sense defined in [12]. Formal relations hold directly between any elements of the model (without an intervening element). Domain-specific formal relations may be introduced in a context model and referred to in SML.

Figure 4 shows an example of a situation type composed of another situation type. The occurrences of composing situation types are represented within the composite situation as nested rounded rectangles in gray. The *SuspiciousParallelLogin* situation type is defined here with two occurrences of situation *LoggedIn* that *overlap* in time, for the same account.

The directed arrow *equals* connecting the bordering diamonds constrains the occurrences of situation *LoggedIn* such that they must include the participation of the same account (regardless of the device used to access the account). These bordering diamonds represent the entities that participate in the situation which are of interest to the composite situation being defined.

The other directed arrow labeled with *overlaps* defines a constraint referring to a temporal formal relation between the situation occurrences, in such way that both occurrences must overlap in time. SML allows composition of situations using the temporal formal relations defined by Allen [1], namely *before*, *meets*, *overlaps*, *starts*, *during*, *finishes*, *coincides,* and their converse relations *(after*, *met by*, *overlapped by*, *started by*, *includes*, and *finished by*).

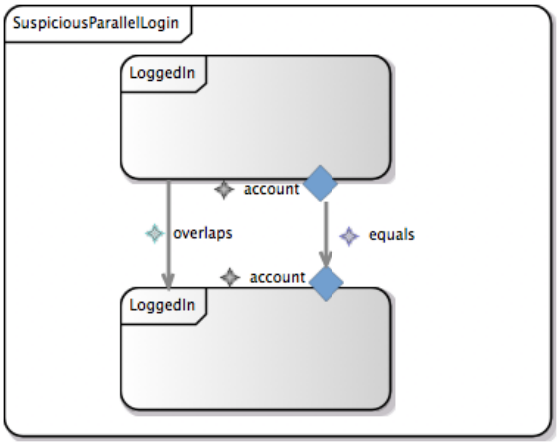

Figure 4. Situation SuspiciousParallelLogin

Figure 5 depicts an example timeline for an instance of *SuspiciousParallelLogin*, in terms of two occurrences of situation *LoggedIn* (for the same account). Note that the situation only exists when both occurrences exist simultaneously.

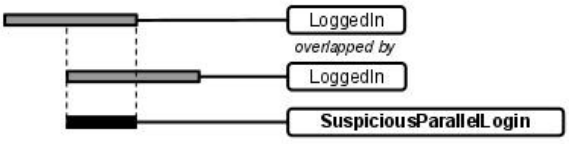

Figure 5. Example timeline for Situation SuspiciousParallelLogin

Figure 6 depicts a more complex situation type called Situation *SuspiciousFarawayLogin*, in which we define additional constraints. The *SuspiciousFarawayLogin* situation type is defined by two *LoggedIn* occurrences (for the same account), in which the first occurrence must have

ceased at most 2 hours earlier than the second. This temporal formal relation is specified by the directed arrow *before*, which is parameterized with lower and upper time limits (between 0 and 2 h). *Past* situations, such as the first occurrence of Situation *LoggedIn*, are graphically represented by nested rounded rectangles in white.

The situation type *SuspiciousFarawayLogin* also prescribes that the *Locations* of the entities of type *Device*, which are participating in the nested situation types, should not be near each other, at the time the respective situations begin to occur. The properties of entities are represented as oval shapes. In this example, the parameterized formal relation *not near* is employed to constrain the two *Locations*, such that they must be at least 500 km apart from each other.

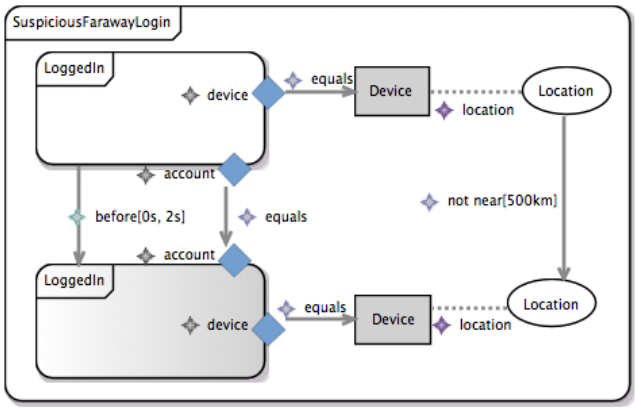

Figure 6. Situation SuspiciousFarawayLogin

Figure 7 depicts an example timeline for an instance of *SuspiciousFarawayLogin*, in terms of two (non-parallel) occurrences of situation *LoggedIn*, for the same account. Note that the situation begins to exist simultaneously to the second occurrence of situation *LoggedIn*, which has started one hour after the first. When the second occurrence of situation *LoggedIn* ceases to exist, so does the occurrence of situation *SuspiciousFarawayLogin*.

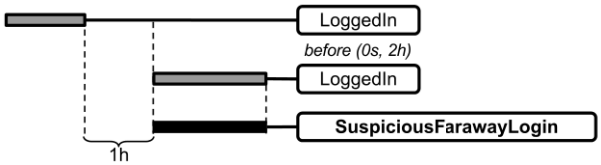

Figure 7. Example timeline for Situation SuspiciousFarawayLogin

SML also allows defining constraints for situation properties (initial and final time, duration, etc.). For example, in Figure 6, we could have made explicit reference to the initial time and final time of the first occurrence of situation *LoggedIn*, instead of using the temporal relation *before*. Figure 8 depicts the situation *AccountUnderObservation* in which we explicitly make reference to the final time of a situation. The situation *AccountUnderObservation* occurs when an account has made *any* suspicious withdrawal in the past 30 days. The icon for the existential quantifier (∃) indicates that any instance of situation *OngoingSuspiciousWithdrawal* for a given account can be matched here, as long as it respects the constraints in the situation type. In this case, the withdrawal must have occurred in the past 30 days, which is represented by the formal relation *within the past* that relates the final time attribute of a situation with the literal 30 days. The 30 days window stretches for more 30 days every time a new Situation *OngoingSuspiciousWidrawal* initiates for a given account. The consequence of this construction is that the account will no longer said to be in the Situation *AccountUnderObservation*, in case no suspicious withdrawals have occurred in the past 30 days.

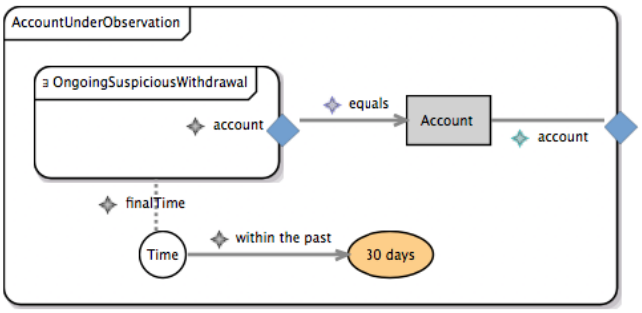

Figure 8. Situation AccountUnderObservation

Figure 9 depicts an example timeline for an instance of *AccountUnderObservation*, in terms of two occurrences of situation *OngoingSuspiciousWithdrawal*, for the same account. The situation begins to exist when the first occurrence of situation *OngoingSuspiciousWithdrawal* ceases to exist. The time window stretches for 30 days after the final time of the latest (in this case second) occurrence of situation *OngoingSuspiciousWithdrawal*.

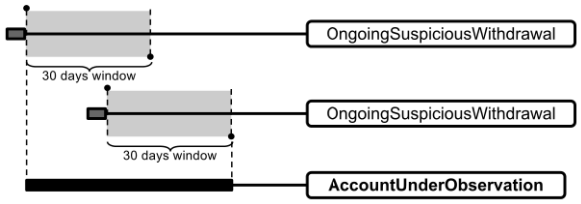

Figure 9. Example timeline for Situation AccountUnderObservation

V. A Formal Semantics for SML

We discuss here a formal semantics for SML. Our purpose is to define the language precisely by defining mapping rules from the language's syntactic elements to an underlying logic framework.

A. Frame-based Model

We assume a Kripke-style frame-based model to define the semantics for the context and situation models. The entities, their context (at instance-level) and their dynamics are represented by a model with the following elements:

- A (non-empty) set W of worlds, with each world *w* representing what exists at a particular point in time.
- A (non-empty) set U of all possible entities and dependent context elements (representing all possible instances of the classes in the context model).
- A (non-empty) set S of all possible situations (instances of situation types in the SML model).
- A binary relation R, representing the accessibility between worlds; accessibility reflects changes from one

point in time to another: the creation and destruction of entities and relational context, as well as the change in value of intrinsic context. (We consider an asymmetric, irreflexive and intransitive relation, representing the direct accessibility between subsequent worlds.)

- A predicate existsInWorld(*w*, *e*) that holds if an entity *e*∈(U ⋃ S) exists in world *w*∈W.
- A function time(*w*) that determines a value in a time structure for a particular world.
- Other predicates and functional symbols derived from the context and SML models (and discussed in the sequel).

Figure 10 illustrates a possible structure of the frame-based model in our banking example. It reveals a particular progression of worlds with two accounts (a1 and a2), one of which (a1) is being accessed by a device (d1) in worlds w1, w2, and w3 (and not in w4). There is a parallel login in w2 and w3, when a1 is accessed by both device d1 (situation s1) and device d2 (situation s2). Direct acessibility is represented by the arrows, thus, R = { <w1,w2>, <w2, w3>, <w3, w4> }. The transition between w1 and w2 is promoted by an occurrence creating the "Access" relational context ac2. The transition between w2 and w3 is the result of the destruction of account a2. The transition between w3 and w4 is the result of the destruction of ac1 and ac2.

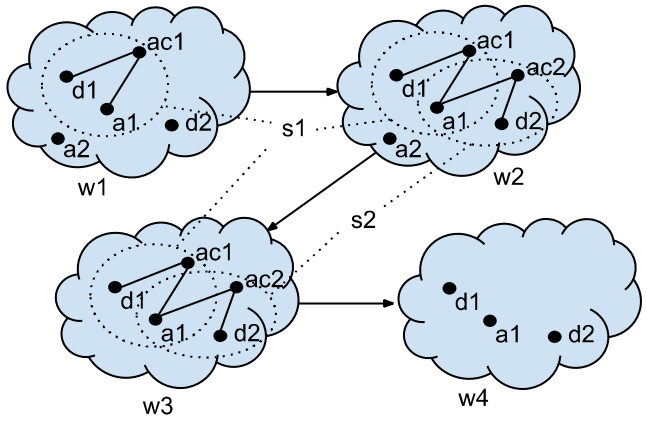

Figure 10. Illustration of the frame-based model

We define a universal predicate for each entity class in the context model, which is true of a particular entity when it is present in world *w* and is an instance of that class in a particular world. In our example, Account(w1, a1) holds, while Account(w4, a2) is false.

We likewise define a universal predicate for each relational context type in the context model. In our example, Access(w1, ac1) holds, while Access(w4, ac1) does not. We also defined a predicate for each association in the context model. In this case, the predicate includes a world and the related elements in its domain (each of the association ends in the class diagram becomes a parameter of this predicate). Thus, in our example, isAccessed(w2, a1, ac1) and isAccessing (w2, d1, ac1) hold, while isAccessed (w4, a1, ac1) and isAccessing (w4, d1, ac1) do not hold.

Finally, we define a function for each intrinsic context type and attribute in the model; the domain of this function includes a world and the entity characterized by the intrinsic context (or attribute). The range is determined by the datatype associated with the intrinsic context type in the context model, thus involving potentially complex values. For example, Location(w1, d1) maps to latitude-longitude coordinates of a device.

B. Simple Situation Types

A simple situation type (such as e.g., *LoggedIn*) is interpreted as an *open sentential formula* formed by a conjunction of terms, each of which corresponds to an element in a situation type. The formula includes a free variable (i.e., parameter) to represent the world *w* in which the situation exists, a free variable to represent the situation itself, as well as a free variable to represent each entity and relational context element in the situation.

Consider the *LoggedIn* situation type; it consists of three participating elements: Device, Account and Access, corresponding to d, a, and ac in the formula. Each of the participating elements in this situation (Device, Account and Access) contributes a term to the formula, using the universal predicate for each type. The connections between entities and context types in the situation type correspond to links and are also added in this conjunction, yielding:

LoggedIn(w, s, d, a, ac) iff
(Device(w, d) ∧ Account(w, a) ∧ Access(w, ac) ∧
isAccessing(w, d, ac) ∧ isAccessed(w, a, ac)).[1]

We define that a situation of a particular type exists in a particular world iff there is a binding in that world which makes the formula true. The definition of the situation type is thus complete by admitting an axiom with the universal closure of the formula corresponding to the situation type (i.e., prefixing the formula with a universal quantifier for each free variable). Formally, for our *LoggedIn* example:

∀w∈W, ∀s∈S, ∀d∈U, ∀a∈U, ∀ac∈U,
LoggedIn(w, s, d, a, ac) iff
(Device(w, d) ∧ Account(w, a) ∧ Access(w, ac) ∧
isAccessing(w, d, ac) ∧ isAccessed(w, a, ac)).

For each situation type, we must also admit an axiom to guarantee that the situation is unique for a particular binding in a particular world. Formally, in our example:

∀w∈W, ∀s∈S, ∀s'∈S, ∀d∈U, ∀a∈U, ∀ac∈U,
((LoggedIn(w, s, d, a, ac) ∧
LoggedIn(w, s', d, a, ac)) → s=s').

Further, if the binding remains stable in subsequent worlds, the situation is also the same. Thus, we also admit the following axiom:

∀w∈W, ∀w'∈W, ∀s∈S, ∀s'∈S, ∀d∈U, ∀a∈U, ∀ac∈U
((w R w') ∧ LoggedIn(w, s, d, a, ac) ∧
LoggedIn(w', s', d, a, ac)) → s=s').

(In our example, this means that the identity of s1 is preserved across w1, w2 and w3; while the identity of s2 is preserved across w2 and w3.)

Constraints on the values of intrinsic context types are also included in the open formula corresponding to the

[1] Note here that we omit from this formula the following terms which are superfluous as all elements in this situation are elements of different types: d≠a, d≠ac, ac≠a; However, these would be required for the case in which two elements of the same type appear as participants in a situation type.

situation type. Each built-in formal relation (such as greater than, not near, equals) is interpreted as a predicate involving potentially complex values. In our example, the OngoingSuspiciousWithdrawal situation type leads to the following formula (where greaterThan(x, y) is defined as x>y):

OngoingSuspiciousWithdrawal(w, s, a, atm, ow) iff
(Account(w, a) ∧ ATM(w, atm) ∧ OngoingWithdrawal(w, ow) ∧
hasWithdrawal(w, ow, a) ∧ doWithdrawal (w, ow, atm) ∧
(greaterThan(OngoingWithdrawalValue(w, ow), 1000)).

As discussed for the case of the *LoggedIn* situation type, we should admit as an axiom the universal closure of the formula above, as well as axioms for uniqueness and presentation of situation identity (similar to those defined for *LoggedIn)*. We omit them here for the sake of brevity, and focus on the construction of the open formula corresponding to each situation type in SML.

C. Complex Situation Types

A complex situation type, i.e., a situation type composed of other situation types, can also be interpreted as an *open sentential formula*. The formula corresponding to a complex situation type is derived using the same rules discussed for simple situation types with the addition of terms for each composing situation, each of which add a free variable to represent the composing situation.

In our example, the *SuspiciousParallelLogin* situation type leads to the following formula:

SuspiciousParallelLogin(w, s, s, s') iff
(LoggedIn(w, s, d, a, ac) ∧
LoggedIn(w, s', d', a', ac') ∧ (s≠s') ∧
overlaps(s, s') ∧ (a=a')).

The overlaps predicate holds whenever *s* and *s'* exist simultaneously and *s* begins before situation *s'*. Formally:

∀s,s'∈S overlaps(s, s') iff
∃w∈W (existInWorld(w,s) ∧ existInWorld(w,s')) ∧
(initialTime(s)<initialTime(s')).

The initial time of a situation is defined formally by:

∀s∈S, ∀w∈W ((initialTime(s)= time(w)) iff (w=firstWorld(s)) iff
existsInWorld(w, s) ∧ ¬∃w'∈W ((w' R w) ∧ existsInWorld(w', s))

While each term that corresponds to a *current* situation uses the variable *w* to refer to the current world (cf. example *SuspiciousParallelLogin* above), each term that corresponds to a *past* situation contributes an additional free variable to represent the past situation and world. This is evident in the *SuspiciousFarwayLogin* example, which leads to the following formula:

SuspiciousFarawayLogin (w, s, s', s'') iff
(LoggedIn(w, s', d, a, ac) ∧
LoggedIn(wpast, s'', d', a', ac') ∧
(s'≠s'') ∧ (w≠wpast) ∧ before(s', s'', 2) ∧ (a=a') ∧ notnear(
Location(firstWorld(s'),d), Location(firstWorld(s''),d'), 500km)).

Here, before(s1, s2, t) is a built-in temporal predicate. Figure 11 shows all supported temporal predicates. Time is represented in the horizontal axis and situations in black represent inactive situations (those that have ended). The formal definitions rely on comparisons of the initial time (shown above) and the converse final time of situations.

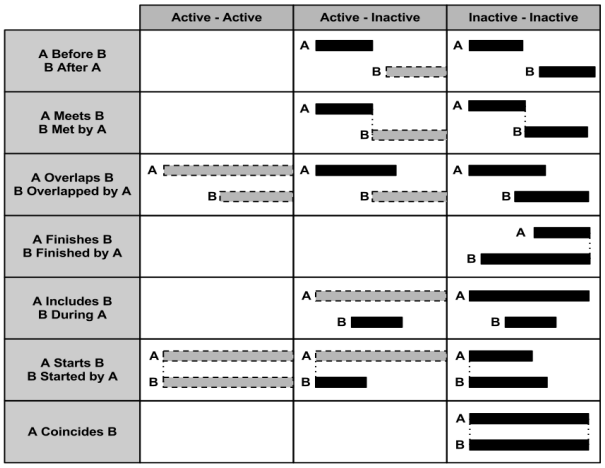

Figure 11. Supported temporal relations between situations

VI. SITUATION DETECTION REALIZATION

Once the required situation types have been specified in SML, we proceed by deriving an implementation for situation detection and handling using as a starting point the SML models. As implementation platform we have chosen the Drools general-purpose rule-based engine due to its performance, availability and community support. Context information is fed as facts into the working memory of the Drools rule engine and rules generated in the transformation of SML models are executed to detect situations.

Since there is no direct support for situations in Drools, we have opted to add functionality on top of the platform to provide such support. This enables us to reflect the overall structure of the SML specification with a simple rule pattern at the implementation level, which is beneficial with respect to traceability and simplicity of the transformation of SML models. We call the additional support for situations in Drools *Drools Situation*. Drools Situation builds on Drools Fusion, which is a Drools module responsible for complex event processing capabilities. No modification of the general-purpose Drools engine is required, and the support for situations is added through general rule patterns for situations and situation-independent helper classes.

Figure 12 depicts the elements of our approach, relating specification level and realization level as well as design time and run-time aspects.

The upper part of the figure shows the design time correspondences between elements of the specification level and the realization level. A UML context model is transformed into Java classes in an Acceleo (http://www.acceleo.org/) transformation presented in [4]. The transformation of SML models leads to Java classes and a situation detection rule set which rely on Drools Situations. We focus here on the design of this transformation (in section VI.A); the implementation of this transformation in Acceleo is presented in [4], along with the EMF metamodels for context and situation models.

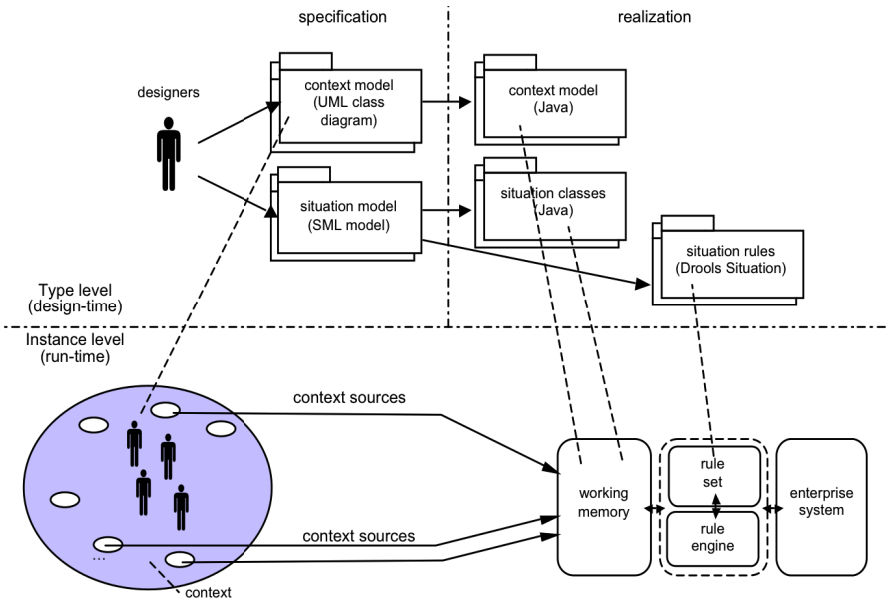

Figure 12 Overview of the model-driven and rule-based approach to situations

The lower part of the figure reveals the run-time relations between the sources of context information and the rule-based implementation. Context sources provide context information, which is input as facts in the engine's working memory. These facts are Java objects instantiated from the Java classes that have been generated from the context model. The rule engine uses the generated situation detection rule set to detect situation instances from the facts in the working memory. The mechanism used for rule application in Drools (and in our case for situation detection) is based on the Rete algorithm (introduced in [11]), which matches the patterns for situations by remembering past pattern matching tests. Only new or modified facts are tested against the rules, which garantees the efficiency of pattern matching.

A. Generation of a Situation Rule

Each situation type corresponds to the definition of a single rule in Drools Situation, which we call *situation rule*. Conditions for firing a situation rule, i.e., the Left-Hand-Side (LHS) of a situation rule, are derived from the situation type elements and constraints. The Right-Hand-Side (RHS) of a situation rule invokes helper functionality of Drools Situation in order to initiate the situation life-cycle management.

Figure 13 shows the situation rule generated from the SML model for Situation *LoggedIn* (Figure 2).

```
rule "LoggedInRule"
when
     account : Account()
     device : Device()
     $access1 : Access(isAccessed == account,
                       isAcessing == device)
then
        SituationHelper.situationDetected(drools,
LoggedInSituation.class);  end
```

Figure 13. LoggedIn situation rule.

The generated LHS of the rule is a conjunction of Drools patterns (lines of the "when" clause) which are derived from the entities, relational contexts and nested situations that participate in that situation. Patterns in a rule contain constraints representing requirements on the entities' context values and relations. The situation rule depicted in Figure 13, for example, includes three patterns in its LHS, representing entity types *Account* and *Device*, and the relational context type *Access*. The pattern corresponding to the relational context *Access* is constrained in such way that it applies to the participating device and account (reflecting the links between the relational context and the entities).

The use of operators in SML generates constraints in the Drools rule. These constraints always include an operator, parameterized or not, and a reference value to be compared to, that can be a literal or a binding variable from another pattern. Figure 14, for example, depicts the code for the *OngoingSuspiciousWithdrawal* situation type (Figure 3), in which the last rule pattern narrows down to the instances of the relational context *OngoingWithdrawal*, whose values are greater than $1000,00.

```
rule "OngoingSuspiciousWithdrawalRule"
when
   $account1 : Account()
   $atm1 : ATM()
   $ongoingwithdrawal1 : OngoingWithdrawal(
                value > new Monetary("$1000"),
                hasWithdrawal == $account1,
                doWithdrawal == $atm1)
then
   SituationHelper.situationDetected(drools,
OngoingSuspiciousWithdrawalSituation.class); end
```

Figure 14. OngoingSuspiciousWithdrawal situation rule.

The Right-Hand-Side (RHS) of the situation rule invokes the *situationDetected* method, which in turn instantiates what we call a situation class and starts situation lifecycle

management. Each situation class corresponds to a situation type in SML. Situation classes are used as situation fact templates, and are simple classes, with an attribute for each entity that plays a role in the situation.

A referred nested situation in SML is also mapped to a Drools pattern, as depicted in Figure 15, which specifies a situation rule for the Situation *SuspiciousParallelLogin* (Figure 4) in terms of two occurrences of the *Situation LoggedIn*. The pattern refers to the situation class of the nested situations. Similarly to other relations, temporal relations are also mapped to constraints. Figure 15 illustrates this for the "overlaps" temporal relation.

For each nested situation, such as the *LoggedIn* situations, we should specify whether they are current (active) or past (deactivated). For that, we include the *exists* operator, which checks whether a *CurrentSituation* fact exists for that situation. Similarly, for past situations, a *not exists* condition for the *CurrentSituation* fact should be defined for that situation. In Section B we further discuss how *CurrentSituation* facts are managed in the working memory (inserted and retracted).

```
rule "SuspiciousParallelLoginRule"
when
    $loggedin1 : LoggedInSituation(
                 $loggedin1_account:account)
    exists(CurrentSituation (
                 situation == $loggedin1))
    $loggedin2 : LoggedInSituation(
                 this overlaps $loggedin1,
                 account == $loggedin1_account)
    exists(CurrentSituation (
                 situation == $loggedin2))
then
    SituationHelper.situationDetected(drools,
SuspiciousParallelLoginSituation.class); end
```

Figure 15. SuspiciousParallelLogin situation rule.

Drools Situation offers operators for temporal reasoning between situations, analogously to Drools Fusion, which supports temporal operators based on Allen's interval algebra operations [1]. However, Drools Fusion exclusively supports temporal relations between events, which must have predefined final times (not the case for situations).

In order to overcome this shortcoming and still use the temporal operators offered by Drools Fusion, we have overridden current operators in such a way that, when applied to situations, they are inferred by considering the application of operators to the events of situation activation and deactivation. The new operators' implementation is rather similar to the original implementation, but goes a step further, by extracting the event of interest from the situation parameter to, finally, evaluate the operators in these events as currently performed in Drools Fusion.

Figure 16 depicts the implementation code for the *Situation AccountUnderObservation* (defined in Figure 8). In order to restrict to past instances of the *Situation OngoingSuspiciousWithdrawal*, the *not exists* operator for the *CurrentSituation* fact is used, as previously discussed.

The existential operator (□) also generates an *exists* condition, which is met when a *DeactivationSituationEvent* for any *OngoingSuspiciousWithdrawal* situation has been inserted in the working memory within the past 30 days. The Drools operator *over window* is used to define a sliding window interval, for which we assign the 30 days value. This sliding window is automatically stretched by Drools when new instances of the *DeactivationSituationEvent* for that situation are created.

```
rule "AccountUnderObservationRule"
when
   account : Account()
   $ongoingsuspiciouswithdrawal1 :
          OngoingSuspiciousWithdrawalSituation(
                                account == account)
   not(exists(CurrentSituation (situation ==
                  $ongoingsuspiciouswithdrawal1)))
   exists(DeactivateSituationEvent(
      situation == $ongoingsuspiciouswithdrawal1)
                           over window:time(30d))
then
    SituationHelper.situationDetected(drools,
AccountUnderObservationSituation.class); end
```

Figure 16. AccountUnderObservation situation rule.

Table 1 shows a summary of the mappings between SML constructs and Drools Situation constructs.

Table 1. Mappings between SML constructs and Drools Constructs

SML Constructs	Drools Constructs
Situation Type	Drools rule (and a Java Class representing the Situation Fact Template)
Entity types, Relational Context types, and nested Situations types	Rule patterns
Intrinsic Context types	Pattern constraints
Relations (formal, temporal and material)	Pattern restrictions (with operators)

B. Situation Management: Under the Hood

A situation fact life cycle consists of its creation, activation, deactivation and destruction. The activation of a situation fact occurs simultaneously to its creation, and the deactivation occurs when the situation rule's condition no longer holds. When the condition at the LHS holds, a situation instance is created; when the condition no longer holds, the situation instance is deactivated. Deactivated situation instances 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 instances.

The situation's lifecycle management strategy strongly relies on a Drools feature called Truth Maintenance System (TMS). The TMS automatically ensures the logical integrity of facts handled by the rule engine. A logical fact exists in the working memory while the conditions of the rule that has inserted it remain true. Thus, the solution we have used consists on a logically inserted fact produced by a situation rule to reflect the situation instance state (existence or nonexistence). This solution has enabled us to detect the activation and deactivation of a situation instance by means of a single rule specification.

The situation logical fact, which we call *CurrentSituation*, is created by the *SituationHelper* class when the conditions of a situation rule are met. Internally, we use a deactivation rule (Figure 17) to manage the deactivation of a situation. The LHS of the deactivation rule checks when the CurrentSituation logical fact for a situation instance is absent (which means the condition of that respective situation rule has turned false and the TMS has automatically removed the *CurrentSituation* logical fact for that situation instance). The RHS updates the related *SituationType* instance state to *non-active* (a past situation).

```
rule "SituationDeactivation"
when
  $sit: SituationType(active == true)
  not (exists CurrentSituation(situation == $sit)
then
  SituationHelper.deactivateSituation(drools,
 (Object) $sit);
end
```

Figure 17. Situation Deactivation rule

VII. Related Work

There are several approaches to situation specification, which have been classified into learning-based or specification-based and reviewed in [25]. In learning-based approaches, situations are identified by using AI learning methods, such as Bayesian Networks and Decision Trees. In specification-based approaches such as the one proposed here, situation types are explicitly defined by capturing expert knowledge in situation specifications.

Many of the specification-based approaches to situation such as, e.g., [24], [15], [8], [5], [13], often specify situations in terms of logical expressions or formal ontologies. Most of these situation modeling approaches are platform-specific or make use of general-purpose languages, such as OWL and OCL. This means that they are not designed to specifically model situation types and, therefore, do not offer primitive situation constructs. In addition, many of these languages still lack expressiveness with respect to situation composition and temporal reasoning.

As discussed in [5] several approaches presented in the literature [17], [14], [23] support the concept of situation as a means of defining particular application's states of affairs. Nevertheless, these approaches usually offer reactive query interfaces instead of detecting situations attentively. The work presented in [14] 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 the design supports event triggers for situation detection, to the best of our knowledge and as reported in [14], this programming model has not been implemented.

An approach that also applies Allen relations to situations is presented in Reigner et al [20]. The authors model what they call a situation network, relating situations through the various Allen relations. They rely on events to progress from situation to situation, and differently from our approach, do not elaborate on the patterns or conditions (constraints on properties, relations and existence of entities) that define a situation type. We could say that a situation type is "opaque" in their approach, which focus on the relations between situations.

In our previous work, some of us have explored the use of general-purpose UML class diagrams with OCL constraints to define the conditions under which situations of a certain type exist [5]. An invariant in that approach is interpreted as the necessary and sufficient condition for a situation to exist. Situations modeled in SML can also be modelled with the technique discussed in that paper, however, the approach suffers from poor usability, as it relies on extensive use of OCL constraints, in particular to make sure that elements participating in a situation are related to each other. This is unnecessary in SML as path connectedness and nesting imply these constraints, which would necessarily have to be stated explicitly in OCL in that approach. This would lead to more verbose models and would require OCL knowledge for situation modeling. The result is that SML models are more compact and more usable than the corresponding general-purpose models in UML and OCL. Our previous work is also different from the work presented here in terms of the implementation patterns, since Jess, differently from Drools does not support the so-called logic facts (and consequently, truth maintenance had to be supported in the transformation itself.)

In our earlier work some of us have also addressed issues involved in a distributed rule-based approach for situation detection (see [5], [7]). In that work, we have explored two distributed scenarios (beyond a centralized approach): (i) distributed detection with multiple engines detecting independent situations and (ii) a distribution scenario with a higher level of distribution assigning parts of the rule detection functionality to different rule engines. First, we should note that the use of SML does not in principle prevents us from adopting these distribution scenarios at the realization level. Approach (i) should be directly feasible with the realization patterns proposed (using Drools Server to connect to remote engines). Nevertheless, approach (ii) relies on further distribution support from the rules platform. In our earlier work this was provided by a distributed extension of Jess (DJess). Similar support is not yet available for Drools; should this support be available in the future, we expect to be able to address approach (ii) by writing an additional transformation from SML.

VIII. Concluding Remarks

In this paper, we have addressed challenges in the explicit support for situations at both situation specification level and realization level. At the specification level, we have enabled the modeling of situation types through a domain-specific graphical language called SML. At the realization level, we have detected situations with a rule-based approach implemented on Drools. We have proposed implementation patterns for situation rules, as well as

provided support for the implementation of complex situations from more primitive situations both at the specification and realization levels (thus preserving the overall specification structure in the realization). Further, we have implemented temporal operators for situations and the implementation level support for situations in Drools can also be employed independently of SML.

An evaluation of the performance of situation detection is ongoing. Nevertheless, due to our previous experiences with the use of a rule-based approach for situation detection (employing Jess) [5] we expect the performance of situation detection to be adequate for most applications. As we have discussed earlier, the algorithms employed in pattern matching are optimized to avoid repeating unnecessary comparisons for conditions that have not been modified, reducing the effort for situation detection.

We have implemented the support for graphical editing of SML models in an Eclipse plug-in using Obeo Designer. The SML diagrams depicted here have been obtained through screenshots of this plug-in. The situation detection rules that correspond to SML models produced in the tool are fully generated using automated transformations implemented in Acceleo, with no manual intervention required in the Drools code (see [4]).

We intend to continue investigating on the expressive power of SML without compromising the simplicity we have achieved in the visual representation of patterns in situation types. One of the potential additions is a textual syntax for writing advanced situation constraints. We believe this can be offered to expert users in scenarios in which the expressiveness of the graphical language shows to be insufficient. Finally, the concrete syntax presented here should be subject to analysis in terms of the guidelines defined in [16].

ACKNOWLEDGMENT

This research is funded by the Brazilian Research Funding Agencies FAPES (grant number 52272362/2011) and CNPq (grants number 483383/2010-4 and 310634/2011-3). Izon Thomas Mielke is supported by FAPES.

REFERENCES

[1] J. F. Allen, “Maintaining knowledge about temporal intervals,” Communications of the ACM, vol. 26, Nov. 1983, pp. 832–843.

[2] M. Bali, Drools JBoss Rules 5.0 Developer’s Guide, Packt Publishing, 2009.

[3] J. Barwise, The Situation In Logic, CSLI Lecture Notes 17, 1989.

[4] P. D. Costa, T. I. Mielke, I. Pereira and J. P. A. Almeida, Realization of a Model-Driven Approach to Situations: Situation Modeling in Ecore and Rule-Based Situation Detection in Drools, Technical Report, Federal University of Espírito Santo, Brazil, 2012. Available at http://nemo.inf.ufes.br/files/sml_tech_report.pdf

[5] P. D. Costa, J. P. A. Almeida, L. F. Pires and M. J. van Sinderen, “Situation Specification and Realization in Rule-Based Context-Aware Applications,” Proc. 7th IFIP Intl’ Conf. Distr. Applications and Interoperable Systems (DAIS’07), Springer, 2007, pp. 32-47.

[6] P. D. Costa, G. Guizzardi, J.P.A. Almeida, L. Ferreira Pires, M. van Sinderen, “Situations in Conceptual Modeling of Context”. Workshop on Vocabularies, Ontologies, and Rules for the Enterprise (VORTE 2006) at IEEE EDOC 2006, IEEE Computer Society Press, 2006.

[7] P. D. Costa, Architectural Support for Context-Aware Applications: From Context Models to Services Platforms, Ph.D. Thesis, University of Twente, 2007.

[8] K. Devlin, “Situation theory and situation semantics,” in Handbook of the History of Logic, vol. 7, J. Woods and D. M. Gabbay, Elsevier, 2006, pp. 601–664.

[9] S. Dobson, R. Sterritt, M. Hinchey, Fulfilling the Vision of Autonomic Computing, IEEE Computer, vol. 43 , no. 1, 2010, 35-41.

[10] M. R. Endsley, “Toward a Theory of Situation Awareness in Dynamic Systems”, Human Factors: The Journal of the Human Factors and Ergonomics Society, vol. 37 no. 1, 1995, pp. 32-64.

[11] C. Forgy, “On the efficient implementation of production systems”, Ph.D. Thesis, Carnegie-Mellon University, 1979.

[12] G. Guizzardi, “Ontological foundations for structural conceptual models,” Ph.D. Thesis, Centre for Telematics and Information Technology, University of Twente, 2005.

[13] D. Heckmann, “Situation Modeling and Smart Context Retrieval with Semantic Web Technology and Conflict Resolution”, MRC 2005, LNAI 3946, pp. 34–47, Springer, 2006.

[14] K. Henricksen and J. Indulska, “A software engineering framework for context-aware pervasive computing,” Proc. 2nd IEEE Conf. on Pervasive Computing and Communications (PerCom 2004), IEEE Press, 2004, pp. 77-86, doi: 10.1109/PERCOM.2004.1276847.

[15] M. M. Kokar, C. J. Matheus and K. Baclawski, “Ontology-based situation awareness,” Information Fusion, vol. 10, Jan, 2009, pp. 83-98, doi: 10.1016/j.inffus.2007.01.004.

[16] D. Moody, “The ‘Physics’ of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering”. IEEE Trans. Softw. Eng. 35, 6, 2009, pp. 756-779. doi:10.1109/TSE.2009.67

[17] X. Hang Wang, D. Qing Zhang, T. Gu, H. Keng Pung, Ontology-Based Context Modeling and Reasoning Using OWL. Proc. 2nd IEEE Annual Conf. on Pervasive Computing and Communications Workshops (PERCOMW04), USA, 2004, pp. 18−22.

[18] B. Heller, H. Herre, Ontological Categories in GOL, Axiomathes 14:71-90 Kluwer Academic Publishers, 2004.

[19] R. Hoehndorf, “Situoid theory, An ontological approach to situation theory”, M.Sc. Thesis, University of Leipzig 2005.

[20] P. Reignier, O. Brdiczka, D. Vaufreydaz, J. L. Crowley, J. Maisonnasse, Context-aware environments: from specification to implementation, Expert Systems, vol. 24, no. 5, 2007, pp. 305–320.

[21] M. Rosemann, J. Recker, Context-aware Process Design Exploring the Extrinsic Drivers for Process Flexibility, Proc. 7th CAISE Workshop on Business Process Modelling, Development, and Support (BPMDS '06), 2006.

[22] O. Saidani, S. Nurcan, Towards Context Aware Business Process Modeling, Proc. 8th CAISE Workshop on Business Process Modeling, Development, and Support (BPMDS’07), 2007.

[23] T. Strang, C. Linnhoff-Popien, and K. Frank, CoOL: A Context Ontology Language to enable Contextual Interoperability. Proc. of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS2003), 2003, pp. 236−247.

[24] S. Yau and J. Liu “Hierarchical Situation Modeling and Reasoning for Pervasive Computing,” 4th IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems and 2nd Intl’ Workshop on Collaborative Computing, Integration, and Assurance. SEUS 2006/WCCIA 2006., pp. 5-10.

[25] J. Ye, S. Dobson and S. McKeever, “Situation identification techniques in pervasive computing: A review,” Pervasive and Mobile Computing, 2011, doi:10.1016/j.pmcj.2011.01.004.