A Rule-Based Platform for Situation Management

Isaac S. A. Pereira, Patrícia Dockhorn Costa, João Paulo A. Almeida

***Abstract*— This paper introduces a platform for situation management, which supports the development of situation-aware applications by offering (i) design artifacts for situation type specification, and (ii) run-time support for situation lifecycle management (situation detection, which may involve composite situation pattern recognition and ultimately situation deactivation). Our approach leverages on JBoss Drools engine (and its integrated Complex Event Processing platform) and enhances its functionality to natively support rule-based situation-awareness. Our platform allows rule-based situation specification (and further situation lifecycle management) by means of a simple rule pattern. We exemplify our situation-based development approach with an application scenario in the public health domain, in which situation types for detecting and monitoring suspicious cases of tuberculosis are specified as situation rules. The specified rules are then deployed and situation detection is managed by the proposed rule-based situation platform.**

***Index Terms* — rule-based systems; situation specification; situation detection; situation reasoning**

I. INTRODUCTION

A central issue in reactive or proactive systems is the ability to bridge the gap between events that occur in the environment and the particular state-of-affairs of interest (aka situations) upon which the system is required to act (or react to). The field of human factors and ergonomics (HF&E) addresses this issue in a human-goal centric approach by means of a well-established concept called *situation awareness* (SA). In [1], Endsley defines SA as "*the perception of the elements in the environment within a volume of time and space, the comprehension of their meaning, and the projection of their status in the near future"*. Endsley also provides a theoretical framework to SA composition, proposing three levels of SA: (i) the perception level; (ii) the comprehension level, and the (iii) projection level. The first level is related to perceiving the status, attributes, and dynamics of relevant elements in the environment; the second level involves the synthesis of situation elements through the processes of pattern recognition, interpretation, and evaluation of what was perceived; and, the third level comprises the ability to project future actions of the elements in the environment. Although Endsley's reference model has been proposed to support humans, we argue that it can be beneficially applied to support the development of *situation-aware* applications.

We argue that the *situation awareness* concept, as referred to by Endsley, should be exploited by reactive or proactive systems, such as *context-aware* applications. Context-awareness focuses on characterizing the user's environment (context) to promote effective interaction between applications and their users by autonomously adapting application's behaviors according to the user's current situation. In the field of context-awareness, Dey, in [4], was one of the first to make efforts in this direction by proposing the *situation abstraction* concept, which is an extension of the *context* concept and refers to a mean *"to determine when relevant entities are in a particular state so they (applications) can take action"*. Therefore, context-aware applications in the sense of [4] could also be considered as *situation-aware* applications.

As discussed by Kokar et al. in [11], "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.

In order to leverage the benefits of the situation abstraction concept in the scope of context-aware application development, proper support is required at design-time (to specify situation types) and run-time (to detect and maintain information about situations). This paper contributes to such support by introducing a situation management infrastructure. We propose *SCENE*, a platform for situation management that leverages on JBoss Drools engine (and its integrated Complex Event Processing platform) and enhances this engine's functionality to natively support rule-based situation-awareness. Our platform allows rule-based situation specification (and further situation lifecycle management) by means of a simple rule pattern. For our purposes, situation management encompasses support for situation type specification, deployment, situation detection (which may involve composite situation pattern recognition) and situation's lifecycle control. According to Endsley's

Isaac S. A. Pereira is a Masters student at the Computer Science Department, Federal University of Espírito Santo (UFES), Av. Fernando Ferrari, s/n, Vitória, ES, Brazil (email: pereira.zc@gmail.com).

Patrícia Dockhorn Costa is an Associate Professor at the Computer Science Department, Federal University of Espírito Santo (UFES), Av. Fernando Ferrari, s/n, Vitória, ES, Brazil (email: pdcosta@inf.ufes.br).

João Paulo A. Almeida is Associate Professor at the Computer Science Department, Federal University of Espírito Santo (UFES), Av. Fernando Ferrari, s/n, Vitória, ES, Brazil (email: jpalmeida@ieee.org).

framework (see Figure 1), our rule-based situation platform provides automated support for the so-called *situation assessment* phase, which comprises part of the perception level and the comprehension level.

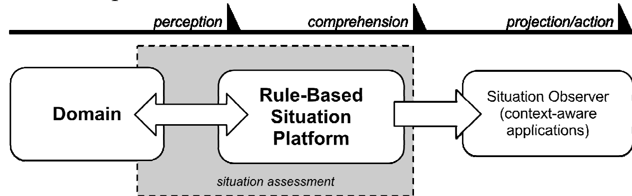

Figure 1. Endsley's reference model and the Rule-Based Situation Platform

The paper is further structured as follows: Section II discusses the notion of situation setting requirements for our approach; Section III presents a motivating application scenario in the public health domain, in which situations for detecting suspicious cases of tuberculosis are specified and managed by the proposed rule-based situation platform (coined *SCENE*); Section IV presents a user's level view of the resulting platform, which is based on the specification of situation types; Section V presents the internal architecture of the platform, discussing the most important implementation decisions; Section VI discusses related work and finally Section VII presents our conclusions and directions for future work.

II. Situations

Situations are composite entities whose constituents are other entities, their properties and the relations in which they are involved [9]. 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 has had an intermittent fever for the past 6 months", "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 [5], [8]), or ascribed an "object" status [11], 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 [11] 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. The situation is said to be *active* while those properties satisfy constraints captured in the situation type. A situation ceases to exist when those properties no longer satisfy the defined constraints. In this case, the situation is said to be a *past situation*. The point in time in which a particular situation instance is detected is called *situation activation instant* and the point in time in which the situation ceases to exist is called *situation deactivation instant*.

Figure 2 provides a graphical representation of the lifecycle of three situations instantiating the same situation type. The vertical axis represents the possible states-of-affairs of the entities in the domain of interest. The horizontal axis represents the passing of time. For the sake of simplicity, suppose we are only concerned with a single property "temperature" of a single entity instance "John" of type "Patient", and we are interested in the situation type "John has fever". This situation type is characterized when John's temperature lies above a given threshold (gray area in Figure 2).

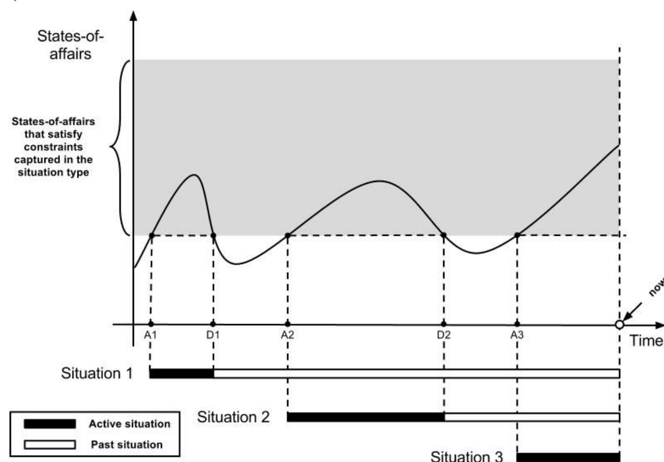

Figure 2. Example of situation instances lifecycle

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. This allows us to consider different levels of situation assessment.

III. Application Scenario

Tuberculosis (TB) is one of the world's most ancient and deadly infectious diseases. According to [16], about 1.4 million people die from TB, and roughly 9 million people develop the disease, each year. One-third of all people on Earth — nearly 2.5 billion people — have a latent form of TB. TB spreads from person to person through air, and its epidemic control lies on accurate diagnosis, drug regimen, and prevention from bacterial exposure over healthy population by infected patients throughout their treatment. Public health programs could benefit from computational systems that help monitoring the population in TB focus areas in order to minimize risks of contagion.

In that sense, we consider an application scenario which comprises the monitoring of particular patients who present a risk of contracting and/or spreading TB, such as: (i) persons who have had recent contact with any infected patients, (ii) persons who have declared to exhibit characteristic *TB symptoms* (clinically confirmed) and even (iii) persons who are already diagnosed and present the *TB disease*. Through a blood test exam called IGRA, a *TB infection* can be detected; however, a positive IGRA does not mean the patient has the *TB disease*, i.e., an infection with symptoms manifestation. A positive IGRA could refer to a *latent infection* case in which TB symptoms are not manifested by the infected patient.

Monitoring patients in groups of risk and determining their current diagnosis situation would be critical for supporting decision making towards a better TB control strategy. In order to allow patient's monitoring, we assume the following contextual data to be available (i) real-time patient's body signals (temperature, blood pressure, heart rate, etc), and also (ii) patient's medical records (previous diseases and past exam results). Based on this contextual information, we define four situation types regarding patients' symptoms and diagnosis:

1. the *TB Infection Situation*, which is considered to exist for every patient whose latest IGRA had a positive result;
2. the *Fever Situation*, which is considered to exist whenever a patient's temperature is above 37º C;
3. the *TB Symptom Situation*, which is considered to exist for every patient presenting a series of recurring *Fever Situations* (intermittent fever). In fact, several other conditions including chills and persistent dry cough are also symptoms of TB. For the sake of clarity in our examples, we have simplified the *TB Symptom Situation* specification to the intermittent fever symptom, only;
4. the *TB Disease Situation*, which is considered to exist for every patient which is simultaneously in the current *TB Infection Situation* and in the *TB Symptom Situation*.

Several actions can be taken upon detection of any of the aforementioned situations. For example, upon detection of a *TB Disease Situation*, the respective patient could be contacted by a health professional; or, upon detection of a *TB Symptom Situation*, a warning SMS could be sent to the patient.

IV. The *SCENE* Platform: User's view

A. Drools

Our approach leverages the *Drools* general-purpose rule-based platform which employs the RETE pattern matching algorithm [6] as a mechanism for rule evaluation (and in our case situation detection). RETE efficiently matches the patterns for situations against *facts* in the Drools Working Memory (WM) by remembering past pattern matching tests. Rules are defined in Drools by means of a domain-specific language called the *Drools Rule Language* (DRL).

A DRL rule declaration comprises a condition and a consequence expression block, respectively referred to as *Left Hand Side* (LHS) and *Right Hand Side* (RHS). A rule specifies that when the particular set of conditions defined in the LHS occurs, the list of actions in the RHS should be executed. The LHS is composed of conditional elements which can be combined through logical operators, such as *and*, *or*, *not* and *exists;* and set operators, such as *contains* and *member of*. A conditional element can be a pattern or a constraint. A pattern matches against a fact in the working memory (of the specified class type); constraints match against properties, and are defined as conditions within a pattern. The RHS allows the declaration of procedural code to be executed when the conditions defined in LHS are satisfied.

B. Situation Specification

In order to address the requirements we have discussed in section II, situation types are specified in *SCENE* by means of structural and behavioral aspects, which are realized by *Situation Classes* and *Situation Rules*, respectively. Every user-defined *Situation Class* specializes the pre-defined class *SituationType*, which is an abstract class that addresses the situation temporal properties and compositional characteristics.

A user-defined Situation Class should structurally define the particular roles played by domain entities in that situation type. For example, consider the Fever Situation type (Figure 3), which is characterized when a person's temperature rises above 37º C. The domain entity Person is playing a role (*febrile*) in the Fever Situation type and should be explicitly defined as such. In our approach, situation properties are tagged as roles using the *@SituationRole* Java annotation. Figure 3 depicts the Fever S*ituation class* declaration in DRL, in which the domain entity Person is tagged as a situation role by means of a *@SituationRole* annotation.

```
1  declare Fever extends SituationType
2     febrile: Person @SituationRole
3  end
```

Figure 3. The *Fever Situation Type* Class declaration

The behavioral part of the situation type specification defines how the abovementioned roles are played in that

particular situation. In order to accomplish that, the roles declared in a *situation type class* are characterized by means of conditional patterns defined in the LHS of the *Situation Rule* declaration. Taking the Fever Situation type example, the Fever Situation Rule (i.e., the behavioral specification, depicted in Figure 4) defines *febrile* as any person whose temperature exceeds 37º C. The role *febrile* is specified as a LHS pattern identifier, which is a binding variable whose value is assigned for each person satisfying that particular condition. By means of these binding variables, we can handle matched facts as objects in the RHS of a rule. Therefore, LHS identifiers are used to handle situation participants, relating them to their respective situation role labels, which should have been previously declared in the *situation type* class. Note that identifiers names should match the property names tagged as situation roles in the Fever Situation class (as defined in Figure 3). *SCENE* uses this information internally to allow proper situation type specification (and further situation lifecycle control).

When a situation rule is fully matched (i.e., the conditions are satisfied), all the facts bound by LHS identifiers that refer to situation roles comprise the so-called *situation cast*. The situation cast is the set of all the entities that participate in the situation (including other situations in composite situation types).

```
1  rule "Fever"
2   @role(situation)
3   @snapshot(on)
4     when
5        $febrile: Person(temperature > 37)
6     then
7        SituationHelper.situationDetected(drools, Fever.class)
8  end
```

Figure 4. The *Fever Situation Rule*

The RHS of a situation rule invokes *SCENE*'s procedural API through the *SituationHelper* module. The invocation of the *situationDetected* method starts the situation lifecycle control (situation creation, activation and deactivation), which is completely realized by the platform. When a situation is activated, a *situation fact* is inserted in the working memory representing that particular situation occurrence.

Situation Rules can also present particular metadata attributes, which are declared before the LHS block. The *@role* metadata is assigned to as *situation* so that the engine can recognize the respective rule as a *situation rule*. The other two metadata attributes are related to what we call *situation's snapshotting* setup. The *situation snapshotting* refers to the process of saving *situation cast* state snaphots throughout the situation's existence. Snapshotting allows composite situation types to constrain past situation occurrences based on situation cast states.

Consider, for example, that we may need to refer to the temperature of John in a particular past occurrence of John's Fever Situation. Since John's temperature most probably changed throughout the active phase of that particular past situation occurrence, a decision should be made about the temperature value to be stored. Therefore, in addition to specifying the need to keep past situation occurrences, *SCENE* allows the specification of three strategies for participation state storage, namely *first*, *stable* and *last*, which are specified in a rule by means of the *@restore* metadata. Table 1 explains in detail the metadata attributes currently supported.

metadata	function
@role	Once tagged with the *situation* value, it allows the situation engine to handle the rule as a *situation rule*.
@snapshot	Turns on snapshotting for the situation cast. It must be turned **on** if the situation type takes part on complex situation compositions, i.e., if the situation type specification refers to past situation occurrences. (In its absence the default is "off".)
@restore	Related to situation composition support. Refers to the participation state storage approach at situation deactivation. It can assume three values: (i) ***first***: sets the participants' state as it was at the situation detection moment (ii) ***stable***: sets the participants' state to the most stable phase throughout the situation's life or (iii) ***last:*** restores the participants' state as it was when the situation was deactivated.

Table 1. Situation Rule Metadata

With respect to the participation state storage strategies, consider, for example, a particular past occurrence of John's Fever Situation in which John's temperature (i) was 38º C at situation activation, (ii) has stabilized in 38,5º C for the longest period of time during situation active state and (iii) was 37º C at situation deactivation. Using the strategies *first*, *stable* and *last*, the following temperature values would be restored, respectively: 38º C, 38,5º C and 37º C. The *@restore* metadata is optional; when omitted, the *stable* strategy is considered as default.

In our TB monitoring scenario, the *TB Symptom* situation is composed of past occurrences of Fever Situation. Therefore, the Fever Situation rule (Figure 4) includes the *@snapshot(on)* metadata in order to turn on its *snapshotting* process. Since no restoring strategy has been defined, the *stable* strategy is considered.

C. Temporal Reasoning

Drools natively provides LHS operators to correlate events in a temporal perspective. All thirteen Allen's operators [2] are supported and also their logical complement (negation). For example, it is possible to define conditions in which an event happens *before* another one, or when both events *overlap* in time (among other possible event correlations). Nevertheless, events in Drools are always records of past occurrences; thus, differently from situations there are no "active" (or current) events. This requires special treatment of temporal operations involving situations, as the final time of active situations is undetermined. We have thus enriched the situation reasoning engine, to allow the definition of constraints for situations using the temporal operators. This allows us to apply temporal operators to pairs of situations, to pairs of events (as supported natively in Drools) and to situation-event pairs.

Figure 5 shows all supported temporal operators. Time is represented in the horizontal direction and situations in black represent inactive situations (those that have ended). Their definitions rely on comparisons of the initial time and the converse final time of situations.

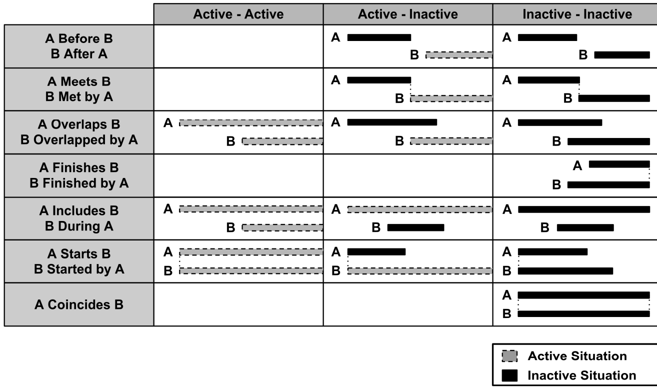

Figure 5. Situation Temporal Relations

The *TB Symptom* situation rule specification (depicted in Figure 6) uses the situation temporal evaluator *after* to describe subsequent episodes of fever of the same patient. The rule's LHS constrains two Fever situations by means of two Fever situation type patterns. The first pattern constrains itself as a past situation (*!active*) and its febrile participant's temperature to be greater than 38.5ºC. Since this pattern refers to a past situation, the participant's restriction considers the temperature values as they were when the situation was occurring. The second pattern constrains itself as a current situation (active) and its febrile participant to be the same as the patient from the first Fever pattern (febrile == *$patient*). In addition, this pattern restricts its own occurrence to be between an hour and a day (*after[1h,2d])* after the past fever occurrence (*$fev1*).

```
1  declare TBSymptom extends SituationType
2     patient: Person @SituationRole
3  end
4
5  rule "TBSymptom"
6  @role(situation)
7     when
8        $fev1: Fever($patient: febrile,
9                     febrile.temperature > 38.5,
10                    !active)
11       $fev2: Fever(this.febrile == $patient,
12                    this after[1h,2d] $fev1,
13                    active)
14    then
15       SituationHelper.situationDetected( drools,
16                                     TBSymptom.class);
17 end
18
```

Figure 6. TB Symptom Situation

Figure 7 depicts an example timeline of an instance of *TBSymptom* situation, in terms of two occurrences of situation Fever, for the same patient. Note that the situation begins to exist simultaneously to the subsequent occurrence of situation *Fever*, which has started two hours after the last one. When the second occurrence of situation *Fever* ceases to exist, so does the occurrence of situation *TBSymptom*.

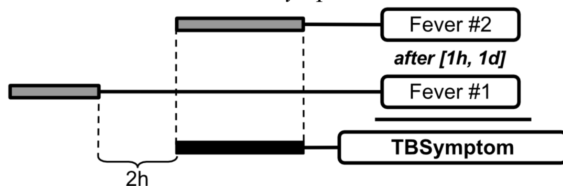

Figure 7. Example timeline for *TBSymptom* Situation

The *TBDisease* (depicted in Figure 8) also involves temporal correlation over situations. It occurs when a patient diagnosed as having *latent TB infection* starts presenting any *TB symptom*. We omit here the specification of the *TBInfection* situation for the sake of brevity, since it requires a simple rule pattern that checks whether the patient has a positive result for the IGRA test. The *during* temporal operator is used to correlate the existence of a *TBSymptom* situation for a particular patient, concurrently to a *TBInfection* situation for the same patient.

```
1  declare TBDisease extends SituationType
2     patient: Person @SituationRole
3  end
4
5  rule "TBDisease"
6  @role(situation)
7     when
8       $tbi: TBInfection($patient: infected,
9                         active)
10      exists(TBSymptom(febrile==$patient,
11                              this during $tbi))
12    then
13     SituationHelper.situationDetected( drools,
14                              TBDisease.class);
15 end
```

Figure 8. TB Disease Situation

Figure 9 depicts an example timeline for an instance of *TBDisease* situation, in terms of a *TBInfection* situation occurrence and any overlapping occurrence of *TBSymptom situation* for the same patient. Note that the *TBDisease* situation begins to exist simultaneously to the first occurrence of *TBSymptom* situation and ceases to exist when the situation *TBInfection* ceases.

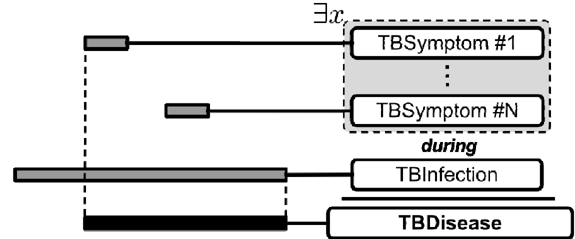

Figure 9. Example timeline for *TBDisease* Situation

Figure 10 depicts an example of reaction rule that is executed upon detection of a TBDisease situation and sends an alert SMS to the patient suggesting to contact healthcare assistance.

```
1  rule "ReactToTBDisease"
2    when
3        TBDisease ($patient: patient,
4                            active)
5    then
6        sendTBAlertSMS ($patient.CelNumber)
7  end
```

Figure 10. Example of reaction rule

V. Situation Platform Realization

A. Situation's lifecycle management

A situation lifecycle consists of situation *detection*, *creation*, *activation* and, possibly, but not necessarily, *deactivation*. As discussed in section IV, situation detection occurs when the LHS of a situation rule is satisfied (for a particular situation cast). Note that conditions in the LHS of a

situation rule hold true while the situation exists (John's temperature exceeds 37º C during the existence of John's Fever Situation). However, although the situation rule may be executed several times while the conditions hold, only one *situation fact* should be created to represent that particular situation occurrence. In order to solve this issue, our approach separates situation detection from its creation.

The situation's lifecycle management strategy benefits from a Drools feature called Truth Maintenance System (TMS). The TMS automatically ensures the logical integrity of facts that are inserted in the working memory in the RHS of a rule. A logical fact exists in the working memory while the conditions (in the LHS) of the rule that has inserted in the working memory remain true, and retracted from the working memory when conditions no longer hold. Thus, the solution we have used consists on a logically inserted fact produced by the firing of a situation rule to reflect the situation fact 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, which otherwise, would require a pair of activation-deactivation rules, as in [7].

Internally, the TMS maintains logical facts by verifying whether there is an equal object already in the working memory before inserting any object. This way, an object only becomes a logical fact in the working memory when it is unique; otherwise, it is discarded by the engine. Therefore, in our approach, a *situation logical object*, which we call *CurrentSituation object*, is created by the *SituationHelper* class for each time the *situationDetected* method is executed. When it is unique (in terms of its situation type and cast), a *CurrentSituation* object is inserted as a *CurrentSituation* fact in the working memory. When snapshotting capabilities are required for a particular situation, *SCENE* keeps serialized versions of the situation cast for each *CurrentSituation* objects creation (i.e., for each execution of the situation rule's RHS). As an example, consider John's Fever Situation. The first time the RHS of Fever situation rule (Figure 4) is executed for a particular cast (which consists of John in this case), a *CurrentSituation* object is created and immediately, through logical insertion, becomes a *CurrentSituation* fact. Further executions of the situation rule's RHS for that particular cast (John) will only produce *CurrentSituation* objects which will be rejected as new facts by the TMS. In our example, serialized versions of *situation casts* referred to by these objects are kept since we have chosen to keep past occurrences of Fever Situation. The TMS automatically retracts John's Fever *CurrentSituation* fact when the LHS of the Situation Fever rule no longer holds for John.

Situation activation occurs simultaneously to its creation, and the deactivation occurs when the situation rule's condition no longer holds. Deactivated situation facts consist of historical records of situation occurrences, which may be used to detect situations that refer to past occurrences.

In order to handle situation activation and deactivation, our approach internally defines a pre-defined pair of rules, which are executed in terms of (existence or nonexistence of) *CurrentSituation* facts. The *Situation Activation* rule (defined in Figure 11) matches for every newly inserted *CurrentSituation* fact (i.e., *CurrentSituation* facts with *situation* attribute set to null). The RHS of the activation rule creates an instance fact of the Situation Type class and its properties are assigned by the corresponding entities involved in that particular *situation cast*. Considering our John's Fever Situation example, when the activation rule is executed, an instance of the Fever class (Figure 3) is created and John is assigned to the attributed febrile. In addition, the *CurrentSituation* fact's attribute *situation* now refers to the newly created situation type instance. This way, the activation rule no longer matches for that particular *CurrentSituation* fact. Upon execution of the situation activation rule, *SCENE* also generates an *initiator* event (*ActivateSituationEvent*), which represents the activation timestamp for that particular situation (and is used for situation temporal reasoning).

```
1  rule "SituationActivation"
2    when
3      $act: CurrentSituation(situation == null,
4                             $type: type,
5                             $castset: castset,
6                             $timestamp: timestamp)
7    then
8      SituationHelper.activateSituation(drools,
9                                        $castset,
10                                       $type,
11                                       $timestamp));
12 end
```

Figure 11. Situation Activation Rule

The *Situation Deactivation* rule (defined in Figure 12) matches for every *SituationType* fact yet active (attribute *active* is true) for which there's no corresponding *CurrentSituation*. The absence of the *CurrentSituation* is a consequence of the TMS logical retraction due to the no longer fulfillment of the situation rule's conditions by a particular *situation cast*. The RHS of the deactivation rule creates a *terminator* event (*DeactivateSituationEvent*) for that particular situation and also sets its transition to a *non active* state (attribute *active* of Situation Type class is set to false).

```
1  rule "SituationDeactivation"
2    when
3     $sit: SituationType(active==true)
4     not(exists CurrentSituation(situation == $sit))
5    then
6      deactivateSituation(drools, (Object) $sit);
7  end
```

Figure 12. Situation Deactivation Rule

B. *Situation Profile Management*

The Situation Profile Manager (SPM) is a module that stores profiles for each situation specification based on declared metadata information previously mentioned in section IV. These profiles allow the situation engine to apply particular management strategies, such as the *cast snapshotting* and *participation state storage* strategies. The SPM assembles rules profiles by parsing the rule base at the execution of the session bootstrapping, capturing situation rule's metadata values. The SPM also maintains the rules profiles throughout the situation's lifecycle execution.

Regarding the lifecycle of *snapshot-enabled* situation facts, the *snapshotting* process takes place for every situation rule's LHS match, in which a serialized version of the assembled *situation cast* (tagged with a timestamp) is stored. When the

situation ceases to hold, the chosen restoring strategy is carried on at situation deactivation (execution of the *deactivateSituation* helper method).

C. *Temporal Evaluators*

The drools native API provides an extensible way to implement new LHS operators. This particular feature allowed us to implement proper evaluators to handle the situation temporal relations, as presented in section IV. Our approach applies the Allen's interval algebra over *initiator* and *terminator* situation events, which are created by the activation and the deactivation rules, respectively (see section V.A).

Given the dynamic nature of a situation occurrence, in which situations may be related to an initiator event only (active situation) or related to both initiator and terminator events (inactive situation), the situation temporal operators have to consider the absence of the *terminator* event.

In order to evaluate situation temporal relations, the situation operators' implementation extracts the events of interest from situations facts parameters and then evaluates the situation temporal relation by means of initiators and terminators events (using the temporal operators currently provided by Drools).

VI. RELATED WORK

There are several approaches to situation specification, which have been classified into learning-based or specification-based and reviewed in [15]. 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., [7], [11], [14], [20], [21], often specify situations in terms of logical expressions or formal ontologies. Most of these situation specification approaches make use of general-purpose languages, such as OWL and OCL. This means that they are not designed to natively support situation specification and, therefore, do not offer primitive situation constructs, such as the ones offered by the proposed rule-based situation platform. Further, in several of these approaches, situation types are reduced to logic propositions, failing to address properties of situations (such as duration) and temporal relations between situations.

As discussed in [7] several approaches presented in the literature [10], [12], [22] 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 [10] 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 [10], this programming model has not been implemented.

In our previous work, some of us have explored the rule-based platform Jess [7] as the foundation for situation detection. Differently from Drools, Jess does not support events, which makes it limiting with respect to temporal reasoning. In addition, Jess also does not support the so-called logic facts, which would require additional support from the situation platform to monitor the situation lifecycle (instead of using native support from the foundation rule-based platform). Using the native support offered by Drools with respect to events and logic facts is beneficial since we expect optimizations to be more efficiently implemented in the core platform.

In our earlier work some of us have also addressed issues involved in a distributed rule-based approach for situation detection (see [7], [9]). 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. 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 adapting the rule-based situation platform accordingly.

VII. CONCLUSIONS AND FUTURE WORK

We have proposed an approach for the specification and realization of situation detection for attentive situation-aware applications. We have implemented a rule-based platform for situation management (coined *SCENE*) that leverages on JBoss Drools engine by adding functionality to natively support rule-based situation-awareness (the source code is available at https://github.com/pereirazc/scene). Situation specification requires a single rule pattern following the standard Drools rule's constraint dialect. Situations can be composed of constraints over domain entities, and in addition can be composed of existing situations themselves. We have addressed the temporal aspects of applications, and included operators to relate situations based on their temporal aspects. The detection is rule-based, and is deployed on mature and efficient rule engine and complex event processing technology available off-the-shelf. The platform manages situations by implementing situations lifecycle control, such as situation activation, state maintenance and deactivation.

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) [7] 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.

In addition to providing infrastructural support for situation detection, we have also explored a graphical language (coined SML) for situation modeling. This work has been reported in [18], in which we present model-driven transformations from SML models into situation rules to be executed on our rule-based situation platform.

For future work we intend to continue working on our rule-based platform improving the ease-of-use aspect without compromising the expressiveness of the situation specification approach. For example, in future versions of the situation platform we intend to allow situation rule specification by means of the rule's LHS only. We also expect the platform to be able to automatically enable the snapshotting process for situation types taking part in others situation compositions.

REFERENCES

[1] M.R. Endsley. Toward a theory of situation awareness in dynamic systems. *Human Factors and Ergonomics Society*, 37:32–64, 1995.

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

[3] M. Bali, Drools JBoss Rules 5.0 Developer's Guide, Packt Pub., 2009.

[4] A.K. Dey, "Understanding and Using Context," Personal and Ubiquitous Computing, vol. 5, no. 1, 2001, pp. 4-7.

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

[6] C. L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem", Artificial Intelligence, vol. 19, 1982.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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. pp. 5-10, doi:10.1109/SEUS-WCCIA.2006.25.

[15] 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.

[16] Trial Signals Major Milestone in Hunt for New TB Drugs, *TB Alliance* [Online] 2012, Available: http://www.tballiance.org//newscenter/view-brief.php?id=1046. (Accessed: 25 October 2012).

[17] E. F. Hill. Jess in Action: Java Rule-Based Systems. Manning Publications Co., Greenwich, CT, USA.

[18] P.D. Costa, I. T. Mielke, I. Pereira, J.P.A. Almeida, "A Model-Driven Approach to Situations: Situation Modeling and Rule-Based Situation Detection". Proc. 16th IEEE Enterprise Distributed Object Computing Conference (EDOC 2012), IEEE Computer Society Press, 2012.

[19] 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.

[20] 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.

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

[22] 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.