Event Condition Expectation (ECE-) Rules for Monitoring Observable Systems

Stefano Bragaglia[1], Federico Chesani[1], Emory Fry[2], Paola Mello[1], Marco Montali[1], and Davide Sottara[1]

[1] DEIS, University of Bologna, Viale Risorgimento, 2
40136 - Bologna, Italy
{name.surname}@unibo.it
[2] Department of Modeling and Simulation, Naval Health Research Center
140 Sylvester Road, San Diego, CA 92126, USA
eafry@gmx.com

Abstract. The standardization and broad adoption of Service Oriented Architectures, Web Services, and Cloud Computing is raising the complexity of ICT systems. Hence, assuring correct system behavior with regard to established design and business constraints is of the utmost importance. Run-time monitoring, where the outcomes of an observed system are continuously checked against what is expected of it, is one possible approach to providing the required oversight.

In this paper, we discuss this notion of rule expectations, their violation and/or fulfillment, and use these concepts to define the concept of an Event-Condition-Expectation (ECE-) rule, a variation of the traditional Event-Condition-Action rule pattern. To demonstrate these concepts, we present extensions to the syntax used by the production rule engine, Drools, and describe their use in a medical case study. The clinical decision support system being developed monitors rule evaluations and expectations, detects constraint violations and is able to take recovery/compensation actions as appropriate.

1 Introduction

In the last ten years there has been a flourishing of models and technologies for developing, deploying, and maintaining ICT systems based on heterogeneous and distributed components. Paradigms such as Service Oriented Architectures (SOA), Web Services (WS), Cloud Computing, Business Process Management Systems and Workflows (BPMS) have been already largely adopted by the ICT industry. When focusing on the medical and healthcare context, automated clinical guidelines [4], care plans, and clinical decision support in general aim to ensure that care standards can be implemented reliably and effectively. These solutions allow for increasingly complex systems, while the adoption of standards pushes for the use of heterogeneous, third-party (software/hardware) components. Consequently, assuring the correct behaviour of such systems is becoming a harder task. Traditional debug techniques alone might not be enough, either

F. Olken et al. (Eds.): RuleML 2011 - America, LNCS 7018, pp. 267–281, 2011.

because of the complexity of the overall system, or because third-party components are often treated as black boxes and so it is not possible to use debugging tools. Run-time monitoring techniques could be of some help by checking if the system behaves correctly while it is *executing*. Typically, a system developer specifies in advance the correct outcomes for any/some possible input. By observing the inputs and the outputs of the system, a monitor automatically verifies that the outputs match with the expected outcomes. In case of a positive answer, the system is deemed to be *conformant*. Such approach is exploited also to verify the correctness of complex systems against some high-level constraints. For example, Quality of Service (QoS) criteria must be continuously monitored, and proper actions must be taken if such criteria are not met. Likewise, legal aspects and business constraints could be subjected to monitoring. A possible way for expressing the desired behaviour of a system is by means of *rules* (ECA Rules, logic-based rules, etc.), that define the *expectations*. Intuitively, we consider an expectation a "justified state of anticipation" about the state of a system, given that some *conditions* on the current system's state and its environment already hold. However, to the best of our knowledge, the notion of *expectation* is rarely used as a "first-class entity" and it is only partially supported. Instead, we strongly believe that it can greatly help in the task of specifying the desired behaviour of a system, since it is a natural, direct way for characterizing the desired behaviour of a system. In addition, we also propose to exploit the concepts of *fulfillment* and *violation* to model the actual outcome of an expectation. Taking inspiration from the ECA rule family, we propose the "Event Condition Expectation (ECE-) Rules". In such rules, the premise defines the facts and events required to trigger an expectation, while the conclusion contains the description of what is expected to be observed. In this paper, we present our idea of expectation and ECE-Rules, and discuss the features that should be supported by this new class of rules (Section 2). We present also a first, prototypical implementation of these concepts based on the Drools [12] Framework (Section 3). In particular, we provide a new language for ECE-Rules (Section 4), and show its application in a real scenario (Section 5).

2 Expectations, Their Fulfillment and Violation

2.1 Related Works

The notion of expectation has been a research subject in many different IT-related fields, where the need for monitoring systems at run-time found a possible solution in the idea of checking the *conformance* of the observed system against some expected behaviour.

The deontic concepts and operators useful to represent norms, obligations and similar concepts have been individuated and mapped to an abductive logic programming context in [2]. Although motivated by a different scope, it provides a background and starting point that is also valid for the semantics of our proposal. In the Multi-Agent Systems (MAS) field, social approaches specify the agents' allowed interactions as expected behaviours (externally observed), and define

fulfillment/violation in terms of deviance from what is expected. The framework SCIFF [1], for example, is mainly focused on a logic-based notion of *expectations* and their fulfillment/violation. Commitments, as deeply investigated by authors such as Singh [17,5,20] or Colombetti and Fornara [7], are defined as promises arising from agents interaction: a debtor agent becomes committed towards a creditor (i.e., it is *expected*) to bring about a certain property, i.e. make it true. In the Business Process research field, van der Aalst and colleagues propose declarative languages that focus on the properties that the system should exhibit: in the DecSerFlow language [14] the users can specify which are the business activities that are (not) *expected* to be executed, as a consequence of previously (not) executed activities. Within the field of legal reasoning and normative systems, authors like Governatori and Rotolo [11,16,9,10,8] have proposed temporal logic frameworks and languages to represent legal contracts between parties: primarily focused on *compliance* issues, such tools simulate the possible course of actions of a system and evaluate if contract agreements are indeed respected.

In the following we introduce our vision about expectations, and discuss a (preliminary, not exhaustive) set of features that, from our viewpoint, should be strictly linked to the notion of expectation, and to the idea of ECE-Rules.

2.2 Desiderata for the Notion of Expectation

Expectations should be about events and/or properties. In the literature it is possible to find expectations about different possible outcome types. An expectation could be about the *happening of an event*, described in terms of the happening time instant and the event duration, or it could be about the reaching a certain state of affairs, or a *property becoming true*, such as in the case of commitments. In particular, it could be desirable that such property is true in a certain instant, or for a whole time interval, continuously [19]. Notice that, in any case, we need to deal with (at least) a description, an initial time instant (a *start*), and a time duration (or, equivalently, an *end*).

Expectations should support closed, as well as open interactions. In *closed-interaction* complex systems, the set of possible events is tipically known a-priori, and at any given time, only few events are supposed/allowed to happen: unexpected events are an evident violation, since anything not explicitly expected is implicitly forbidden. However, the ongoing challenges posed by the new Internet era require very "flexible" systems able to cope with unpredictable, changing and dynamic environments, where the autonomy of interacting entities and the possibility of exploiting new opportunities must be preserved as much as possible [21], or the expertise of workers must be exploited at best [15]. In this context, "closed approaches" show some drawbacks in this regard. *Open interactions* overcome such limits, since they focus on the (minimal) set of constraints that must be guaranteed to successfully interact. To this aim, a further step is required: expectations about what should *not* happen should be supported, thus involving some notion of *prohibition*.

Expectations should be about future, as well as past events. Expectations should allow the user to specify *when* a certain event is expected. It is very intuitive to define expectations about future events, but it is also mandatory to be able to treat past expectations. Within a monitoring setting, it is often assumed that at any time instant T the set of already happened events is fixed, i.e., that any newly observed event will happen at a time greater than T (with a possible approximation Δ). Determining if an expectation about a past event is satisfied or not amounts to look for a matching event in such fixed set. Differently, any reasoning on expectations about future events should be "suspended", as far as future events are unknown. W.r.t. the time axis, monitoring is usually *closed* towards the past, and *open* towards the future.

Expectations should support temporal deadlines. In general, without any temporal deadline, it would not be possible to determine if an expectation has been fulfilled or violated. The problem is somehow mitigated for expectations about past events due to the the semi-open nature of the monitoring task, while any conclusion would be indefinitely suspended by an ideal monitor for expectations about future events. To avoid such situations, some systems (e.g., [1]) envisage a special *closure* event, whose semantic is that no more events will ever happen, allowing the monitor to draw final conclusions.

2.3 Desiderata for a Framework Supporting Expectations

Support for one/many matching functions. When an event happens, it is necessary to decide if it has any relation with what is currently expected. This task is performed by a matching function, that decide if there is a compatibility between the two. A simple matching function could evaluate the syntactical equality of the events' descriptions. However, only more powerful matching functions based on, for example, regular expressions, fuzzy pattern matching, or Prolog-like unification, would be of practical interest. Ideally, a framework should allow for many different matching functions, and, possibly, also for custom-defined functions.

Support for fulfillment and violation. It should be always possible to know if an expectation has been satisfied (fulfilled) or not (violated). Note that, in case of open-interaction systems, such notions would apply differently to expectations and to prohibitions. Roughly speaking, in the former case an expectation could be considered fulfilled if at least a matching event happens, while in the latter the expectation about the non-happening is fulfilled if no matching event ever happened. A doubt might arise about the happening of multiple events matching the same expectation. A framework should support the user to define these characteristics of the fulfillment/violation process.

Support for the life-cycle of an expectation. The life cycle of an expectation should capture all its possible states within the system's life. In particular, we envisage at least three fundamental states: a *pending* one, indicating that an expectation is active but lacking information for deciding its fulfillment or

violation, a *fulfilled* state and a *violated* one. Such states should be accessible as explicit meta-information, both about individual expectations and groups thereof, possibly using them to trigger new events and/or derived expectations.

Support for some notion of "global conformance". If the notions of fulfillment and violation are referred to a single expectation, a framework should support also some global notion of fulfillment (conformance of the whole monitored system). Such notion could be simply defined as a logical "and" of all expectations' fulfillments, or could be user-defined by means of some metrics, such as statistics, or by means of some other criteria like a fuzzy evaluation.

2.4 Desiderata for the ECE-Rules

Direct support for compensation and awarding mechanisms. In our view, ECE-Rules define the events the conditions that trigger an expectation, i.e. make it pending, and resolve it either by fulfillment or violation. Quite often, in the case of violations (res. fulfillments), compensation (resp. award) mechanisms are required: their preconditions can be easily expressed if the framework supports some reification (in terms of events) of the state transitions during an expectation's life cycle. However, such situation is so frequent that also we foresee the possibility of defining compensation/awarding mechanisms directly in the ECE-Rules. So, the rules would be single yet complete "knowledge items", containing the events and the conditions for expectations to become pending, as well as what to do in case the expectation is fulfilled or violated.

3 The Drools Rule Engine

Drools[1] is an open source *"knowledge modelling and business logic integration suite"* composed of several modules. Its core component is a reactive production rule engine, based on an object-oriented version of the RETE algorithm [6].

From a user perspective, the system offers a blackboard-like container, called *Working Memory* (WM), where the facts describing the "state of the world" can be `inserted`, `updated` or `retracted`. The rules, then, are *activated* accordingly whenever the WM is modified. A rule is an `IF-THEN` like construct, composed of a premise (Left Hand Side, LHS for short) and a consequence (Right Hand Side, RHS). The LHS part is composed by one or more *patterns*, which must be matched by one or more facts in the WM for the rule to become active. An active rule is then eligible to be fired, executing the actions defined in the RHS, which may either be logical actions on the WM or side effects. A pattern is a sequence of constraints that a fact must satisfy in order to *match* it. Since facts are objects in Drools, the first constraint is a *class* constraint and the following are boolean expressions involving one or more object's *fields*. The constraints of a pattern are evaluated in chain, acting as a series of increasingly fine-grained filters that progressively retain less and less objects until only the ones matching

[1] `http://www.jboss.org/drools`

```
declare Message @role(event) end
rule "CEP Rule Example"
when
  $m: Message( $s: sender, $r: receiver, content == "HELO" )
      not Message( sender == $r, receiver == $s,
                   content == "+ACK", this after[0,5s] $m  )
then
  log("Acknowledgement expected, not received in 5 seconds");
end
```

Fig. 1. A simple Drools Fusion theory catching the violations of a simple protocol

the pattern are left. Such operation is optimized by the RETE algorithm, which allows to share common sequences of constraints between rules.

The additional modules that Drools provides tackle specific needs such as workflow management, rule authoring and planning: in particular, the module dedicated to *complex event processing* (*CEP*) [13] is called Drools Fusion. In this context, an event is *"a significant state change in an observed system, taking place at a specific point in time"*. Drools Fusion allows to annotate fact classes with metadata: the engine will then consider their instances as event payloads, managing any related temporal information autonomously. Such information may be drawn directly from the objects' fields, or automatically added by the engine, wrapping the instances using appropriate decorators. Drools Fusion adheres to the well-established semantics of temporal intervals proposed by Allen [3]. The engine is not only able to evaluate temporal constraints, but also to delay and schedule activations, and to perform temporal truth maintenance by retracting events irrelevant to the computation. To this aim, the engine has an explicit notion of time maintained by a pluggable clock, which allows to determine the events that fall in a given time interval. The provided default implementations, *pseudo-clock* and *real-time clock*, can be used to control the flow of time during simulations or to build online reactive systems.

As an example, consider the code snippet presented in Fig. 1, which shows how Drools Fusion handles `Message` events to notify the violation of a simple protocol. This protocol requires a server to send back an acknowledge notification within 5 seconds from the moment it receives an hand-shake request from a client. In practice, each time a client sends a `"HELO"` message to a server and no `"+ACK"` message is returned in time, a notification of the violation is logged.

4 The "`expect`" Extension to the Drools Language

In our vision, we would like to exploit a CEP engine to model and solve online monitoring and conformance verification problems. This task would be dramatically simplified if the target system supports the concept of expected behaviour natively. To the best of our knowledge, however, no existing CEP engine provides high-level primitives to handle expectations as discussed in Section 2.

⟨*rhs content*⟩ ::= 'then', { ⟨*expectation block*⟩ }, { ⟨*java statement*⟩ };
⟨*expectation block*⟩ ::= ⟨*expectation list*⟩, { 'or', ⟨*expectation list*⟩ };
⟨*expectation list*⟩ ::= ⟨*expectation*⟩, { 'and', ⟨*expectation*⟩ };
⟨*expectation*⟩ ::= [⟨*id*⟩, ':'], 'expect', ['not' | ['one'], ⟨*id*⟩, ':'], ⟨*pattern*⟩, ⟨*follow-up*⟩;
⟨*follow-up*⟩ ::= ['on', 'fulfillment', ⟨*fulfillment block*⟩], ['on', 'violation', ⟨*violation block*⟩];
⟨*fulfillment block*⟩ ::= '{', { ⟨*repair*⟩ }, ⟨*rhs content*⟩, '}';
⟨*violation block*⟩ ::= '{', { ⟨*repair*⟩ }, ⟨*rhs content*⟩, '}';
⟨*repair*⟩ ::= 'repair', ⟨*id*⟩, ';';

Fig. 2. The ECE sub-grammar, in EBNF form

To this end, we have created an extension to Drools Fusion that enriches the expressiveness of its language with ECE concepts. Such extension is in charge of managing expectations and autonomously executing the proper actions in case of fulfillment, violation or compensation, according to the policies encoded in the rules. Thus the system is no more limited to the traditional Drools' logical operations (`insert`, `retract`, `update`) or free-form side effects, but includes the explicit declaration of expectations.

In our proposed extension, expectations can be nested, so that complex norms may be atomically expressed within a single rule. A nested (conditional) expectation is an expectation generated as a consequence of a previous expectation's fulfillment or violation, and possibly it may depend on the facts or events which triggered the parent expectation.

The ECE rules also automatically generate standard events which reify an expectation and its life cycle, allowing the user to predicate on them using other rules. Although not required in general – and indeed not even recommended is some cases (such as for hard-time constrained CEP), the reification allows the rules to target those concepts directly, possibly even combining them with domain events. An organization that certifies the compliance of some domain's objects to a given regulation, for example, may want to keep track of the amount of violations over the number of activations of a specific expectation, to determine whether some norms need a clearer explanation. Although enabled by default, this feature is actually optional and can be disabled, providing a more slender but slightly less expressive tool.

4.1 The ECE Language

To support ECE-Rules, we have extended the standard Drools parser, allowing the engine to recognize and process the new concepts. Normal rules are handled as usual, while ECE statements are intercepted and redirected to a separate sub-parser and compiler for further interpretation and rewriting. Fig. 2, in particular, contains the extension proposal for ECE-Rules to the Drools language, expressed in EBNF syntax. This grammar shows how to change the consequent of a Drools' rule to include (possibly nested) complex formulas involving

```
rule "ECE-Rule Example"
when
  $m: Message($s: sender, $r: receiver, content == "HELO")
then
  $e1: expect one Message( sender == $r, receiver == $s,
                           content == "+ACK", this after[0, 10s] $m )
    on fulfillment {
      insert(new Message($s, $r, "MAIL")); }
    on violation {
      $e2: expect Message( sender == $r, receiver == $s,
                           content == "+RDY", this after[0, 2m] $m )
        on fulfillment {
           repair $e1;
           insert(new Message($s, $r, "MAIL")); }
        on violation {
           insert(new Message($s, $r, "STOP")); }   }
end
```

Fig. 3. An example of an ECE-Rule

expectations. Any individual expectation has two optional blocks, namely a fulfillment and a violation block, where it is possible to generate new expectations as well as applying consequences. Notice that the ⟨*rhs content*⟩ grammar rule overrides the original one, adding the expectation block; ⟨*java statement*⟩, ⟨*id*⟩ and ⟨*pattern*⟩ are imported directly from the Drools grammar.

Moreover, it is possible to explicitly compensate any previously violated expectation. We have deliberately introduced this feature to allow the recovery of a violation by means of a `repair` statement. This action can be executed as a consequence either of a fulfillment or a violation of an expectation. The only constraint is that the repaired expectation must be defined within the same rule.

To fully understand the scope of our proposal, let us consider the example presented in Fig. 3, which shows the compensation of the violation of a simple protocol, involving a single ECE-Rule. This theory uses the same `Message` class introduced before (not present here for lack of space), but defines a sightly more complex mail protocol[2]. The interaction is again started by a `"HELO"` message *$m*, and a `"+ACK"` messege is expected in 10 seconds: such expectation is labeled as *$e1*. In case of success, the interaction continues with a `"MAIL"` from the client, requesting the list of new mails to the mail server. In case of violation, we proceed instead to the evaluation of the nested expectation: if busy for some reason, the server may still issue a `"+RDY"` message within 2 minutes from *$m*. The rule states that *$e1* may be considered repaired on fulfillment of *$e2*, meaning that it is no more treated as a violation when validating the protocol. In case of a second violation, the client simply closes the interaction with a `"STOP"` message. Notice that the declaration of *$e1* includes the keyword `one`, meaning that, for each message matching *$m*, the fulfillment (violation) branch is only evaluated once after the first `"+ACK"` message is (not) received in time.

[2] In our toy example, any exchange of messages between the client and the mail server (and vice-versa) is accomplished by the insertion of a proper instance of `Message` into the WM.

4.2 The Expectation Meta-Model

To allow the engine to process the grammar defined in section 4.1 and manage the expectations and their life-cycle, we have defined a theory composed by a set of general rules and a set of declared fact classes, whose instances can be manipulated within the WM as appropriate.

Those rules translate any ECE statements into common rules and, optionally, instances of the classes mentioned below, both decorating the starting theory. These rules (not provided for lack of space) permit complex expectations to be spanned over several rules, retaining conceptual dependencies and other relations between expectations, and allow the natural triggering of expectations within other expectations. The straightforward conversion of ECE statements into instances of a few coded class types is often referred as "*reification*": it is not mandatory for the approach, but it allows Drools (that natively works on objects) to declaratively predicate on them. Thus, despite being not the only possible strategy to encode the knowledge, it is very simple and sound, and grants additional expressiveness to the framework. This process is also transparent (the conversion is done autonomously) and optional (it can be disabled to trade some expressiveness for a speedup).

Specifically, the `Expectation` is an abstraction where the (temporal) constraints to be satisfyed are held. The temporal constraints may refer either to the past and the future and have a duration, autonomously handled using Drools Fusion, that starts from the moment they are generated and lasts until the moment they are first fulfilled or violated. `Expectation`s are also identified by a `Label` whose purpose is to group together expectations that refer to the same object tuple, since an `Expectation` can be generated several times, as different object tuples `Activate` the rule at different times, in different `Context`s. A `Context` is usually the initial activation of an ECE-Rule or, in case of nested expectations, the activation of another parent expectation. When an `Expectation` is generated, it is considered `Pending` until it is either `Fulfilled` by an object matching the pattern defined by the expectation, `Violated`, or `Closed`. By aggregating all the `Expectation`s generated within the same `Context`, it is possible to define if the activation of an ECE-Rule has been a `Success` or a `Failure` given the monitoring constraints. For more details on pending expectations and possible outcomes of an expectation see the proper paragraphs in Section 4.3. Eventually, violated `Expectation`s may be `Compensated` by other events. Notice that all the class types introduced here are considered by our system as instant events with the only exception of `Expectation`s whose duration equals their life-time.

4.3 Rule Generation

In addition to the initial, user-provided theory, we preprocess the ECE-Rules and rewrite each one into one or more traditional rules, thus allowing the use of the standard Drools Fusion engine to execute them. Consider an ECE-Rule R such as the one in Fig. 3, having a standard Drools' LHS and an extended RHS

with nested expectations. After parsing the rule, we obtain an *Abstract Syntax Tree* (*AST*) with a regular and recursive structure, derived from the grammar in Fig. 2. Any node i of the AST obtained from an expectation block's expression is the child of a pattern P^i and the ancestor of an `and`/`or` subtree whose leafes are $n(i) \geq 0$ expectations $E^i_{j:1..n(i)}$. Each leaf expectation E^i_j defines an expectation pattern P^i_j and up to two expectation blocks, corresponding to the optional fulfillment and violation branches. Moreover, an expectation block also has a sequence of actions A_i. According to the natural semantics of the language, actions A_i should be executed only when the pattern P^i has been matched. Notice that, in case of nested expectations, any pattern associated to a more external block must have been likewise already matched. In addition, when and only when P^i has been matched, any expectation E^i_j should be generated so that, according to the presence or absence of a fact/event matching its pattern P^i_j, that expectation E^i_j can be considered pending until fulfilled or violated. For every ECE-Rule, the AST thus obtained is visited several times, to create various derived rules which manage the expectations' life cycle.

(Re)action rules. The goal of these derived rules is the execution of the actions A_i when the appropriate conditions apply. One rule is created when visiting the AST for each action block A_i, starting from its root node and using a recursively built LHS L_i, after being initialized with the LHS of the original ECE-Rule R. The first rule is simply `when` L_0 `then` A_0 and corresponds to the rule R *without* the extended expectation block. The action rules are derived only from the simple expectations, so the visitor ignores any `and`/`or` node to process only the expectations E^i_j. When an expectation node with children `on fulfill` or `on violation` branches is encountered, the local positive and negative LHS, L^{i+}_j and L^{i-}_j, are respectively built from the current LHS L_i and the local expectation pattern P^i_j. In particular, the pattern is added in positive form when visiting `on fulfill` children branches, and in negative form, using the negated existential quantifier `not`, when visiting `on violation` branches, i.e. $L^{i+}_j = L_i \cup P_j$ and $L^{i-}_j = L_i \cup \neg P_j$. The resulting rules, up to two for every expectation, are traditional Drools/Fusion rules with no notion of expectation or any other related fact. While generated using a more declarative language, they correspond to the rules which would have been written using a naive, hard-coded approach. The main advantage is that they do not incur in any overhead from the meta-level framework, but still capture the behaviour defined by the business author.

Fulfillment/Violation rules. In addition to executing the business logic, we also want to provide an explicit and automatic management of the expectations' lifecycle, as motivated in paragraph 2.3. To this end, we again visit the AST and generate up to three different rules. The first rule, created visiting a pattern node P_i is used to generate the actual `Expectation` in the working memory. The premise of this rule coincides with the local LHS L_i created recursively during the visit to generate the business rules, while the consequence generates an `Expectation` instance for each expectation node E^i_j defined in the expectation block of P_i. At runtime, when the premise L_i is matched by a tuple T, one or more

expectations will be inserted within the working memory: their context will be associated to the initial activation of the ECE-Rule which triggered the sequence of expectations, but T will be the actual trigger tuple for all the expectations. As soon as an `Expectation` is inserted in the working memory, a `Pending` fact is also generated to trace the fact that the expectation has not been resolved.

In order to close an expectation E_j^i, two additional rules are written. Since the expectations are defined in terms of a pattern P_j^i, which needs to be captured to determine the expectation outcome, the LHS of these rules is simply derived from L_j^+ and L_j^-, adding a pattern matching the appropriate unresolved expectation (i.e. with an existing joint `Pending` fact). The join is performed on the expectation label (known at compile time), the context, and using the bindings on the patterns in L_i to recreate the parent tuple which generated the expectation and use it as a constraint. Notice that, since the LHS of these conformance rules extends the LHS of a corresponding business rule, the RETE will eventually merge them again, eliminating any redundancy introduced at this level. These latter rules are then simply used to instantiate the `Fulfillment` and, respectively, `Violation` events, collecting the required information from the rule activation records, and inserting them in the working memory.

With this pattern-based approach, it is possible that more than one object's pattern P_j^i matches an expectation E_j^i and fulfills it. In fact, expectations remain pending until explicitly retracted or *closured* (if no temporal constraint is imposed). On the other hand, the same fact/event can match different expectations at the same time. If the user wants an expectation to be fulfilled only once (using the `expect one` syntax), the fulfillment rule will be slightly modified to retract the `Pending` fact associated to the expectation, preventing further fulfillments.

Repair rules. ECE-Rules support the "repairing" of violated expectations: for every `repair` statement, a rule is created with the same LHS as its enclosing expectation block. This rule inserts a `Compensation` fact which is joined to its matching `Violation` by a general rule: the `Violation`, then, is marked as compensated.

Success/Failure rules. While fulfillments and violations are handled at the level of each single expectation, we also provide an overall evaluation of the system behaviour in the context of each activation of an ECE-Rule r, as discussed in paragraph 2.3. We provide support for "global" conformance at the level of activations, by expressing and evaluating possibly complex and/or formulas involving expectations. In particular, we define the notion of `Success` and `Failure` from the `Fulfillment` and `Violation` of individual expectations. For complex expectations, the definition is as follows:

$$Succ(E_1 \wedge E_2) = Succ(E_1) \wedge Succ(E_2) \qquad Succ(E_1 \vee E_2) = Succ(E_1) \vee Succ(E_2)$$
$$Fail(E_1 \wedge E_2) = Fail(E_1) \vee Fail(E_2) \qquad Fail(E_1 \vee E_2) = Fail(E_1) \wedge Fail(E_2)$$

When defining success and failure for individual expectations, instead, one must consider any nested sub-expectation, if present, so $Succ(E)$ (resp. $Fail(E)$) is not trivially $Fulf(E)$ (resp. $Viol(E)$). Expectations defined in `on fulfill` (resp. `on`

`violation`) blocks will be generated only when the parent expectation is fulfilled (resp. violated), but they must be taken in consideration when present. Notice that, since violations can be repaired, even expectations generated by violations are still relevant for the final evaluation.

The `Success` of an `Expectation` E is then determined by its fulfillment and either (i) the success of any `on fulfill` expectation block (if E was fulfilled) or (ii) the success of any `on violation` expectation block (if E was violated and then repaired). Likewise, the failure of a parent expectation may be due to its violation, the failure of its fulfillment block or the failure of its violation block (assuming it was violated and then repaired). Notice also that a repaired violation is equivalent to a fulfillment during the evaluation of the overall success/failure, while a non-repaired violation corresponds to the negated form of a fulfillment. Those definitions, then, can be further simplified and expressed as follows by adopting an compact EBNF-like notation:

$$\begin{aligned} Succ(E) &= Fulf(E)\Big[\wedge\Big((Fulf(E)\wedge Succ(E_F))\Big|(Viol^{\star}(E)\wedge Fulf(E_V))\Big)\Big] \\ &= Fulf(E)\Big[\wedge\Big(Succ(E_F)\Big|Fulf(E_V)\Big)\Big] \\ Fail(E) &= Viol(E)\Big[\vee\Big((Fulf(E)\wedge Fail(E_F))\Big|(Viol^{\star}(E)\wedge Fail(E_V))\Big)\Big] \\ &= Viol(E)\Big[\vee\Big(Viol(E_F)\Big|Fail(E_V)\Big)\Big] \end{aligned}$$

Such definitions can be applied directly to the AST tree, generating two conformance rules for each ECE-Rule R that lead to either a `Success` or a `Failure` events for each activation (or nothing if some expectations are still pending). While not *global* per se, such events ease the creation of conformance rules by providing more complex events than fulfillments and violations.

Closure. As outlined in Section 2.2, we regard the expectations as closed toward the past and open toward the future. This choice may lead to some issues when dealing with the expectations that are still pending. The open expectations, in fact, are not caught –and hence not reported– by the monitoring framework (i.e. when halting the system). The procedure of identification and notification of the pending expectations with respect to the open time horizon is called "*closure*" and it is directly managed by our implementation within the general theory. To this aim, a dedicated special fact is inserted into the WM, making a rule to react by joining that fact with any open expectation. As a result, the expectation is closed (its `Closed` field is changed) by flagging a `Violation` (for positive expectations) or a `Fulfillment` (for negative ones). Notice that the clipping of the pending expectations forces the generation of a `Success`/`Failure` event. Our implementation also supports a finer "*targeted closure*": only open expectations whose label matches a specific value are closed.

5 A Use Case in the Medical Field

The Knowledge Management Research (KMR) team at Naval Health Research Center, San Diego has been particularly interested in developing the functional

```
rule "Risk factor evaluation"
when
    $pat : Patient( ... ) $prov : Provider( ... )
    // model result: risk factor and confidence degree
    $risk : HasRisk( $pat, $disease, $factor, $conf )
then
    expect HasRisk( this == $risk, confidence > C_THOLD )
    on fulfillment {       // prediction is reliable
       expect HasRisk( this == $risk, factor < R_THOLD )
         on fulfillment { log( $pat + " safe" ); }
         on violation { /* manage high risk patient */ } }
    on violation {        //request info from patient
        insert( new Message($prov, $pat, "quest" ) ); }
end
rule "Fill Questionnaire Request Protocol"
when
    $m : Message( $prov, $pat, "quest" )
then
    $e : expect Message( $pat, $prov, $answers ; this after[0,T] $m )
        on fulfillment { insert($answers); }
        on violation {
            // T2 > T, giving additional time to $pat
            expect Message( $pat, $prov, $answers ; this after[0,T2] $m )
               on fulfillment { repair $e; insert($answers); }
               on violation { alert( ... ); }
            insert( new SMS($prov, $pat, "quest") ); }
end
```

Fig. 4. PTSD Risk Factor (abstract) Use Case

ontology and semantics required to define "operational constraints" to the execution of a rule. Operational constraints are supervisory (meta) rules that establish expectations for the clinical context a given rule was designed for – they help ensure that the resultant events or behaviors are appropriate for the setting. For example, they might modulate the recommendation for a blood transfusion in a trauma case when the patient "context" is that of a Jehovah's Witness.

Separating the logic of a discreet medical decision from that used by operational rules used to manage clinical context has important implications. Decision support rules should be authored with a focused clinical perspective and that the final result is dependent on other clearly defined orthogonal perspectives that the knowledge management system must orchestrate. It implies that rule execution is not necessarily an all-or-none event, but rather a cascade of constraint evaluations that in aggregate ensure the system's end behavior is individualized and nuanced. This separation of concerns helps clarify what is evidence-based best clinical practice and what are non-medical restrictions imposed on the delivery of that care by the realities of the real-world and the individual patient. As an example use case, consider the following scenario. After returning from a yearlong deployment, a United States Marine is seen by his physician. The doctor's Clinical Decision Support System (CDSS) collects all relevant information about him and feeds it into a Post Traumatic Stress Disorder (PTSD) predictive model that estimates he has a 35% risk of developing PTSD within the next three years. Recognizing, however, that several important historical facts about his past medical history are missing, the patient is asked to take an online survey at home. When he forgets to complete the survey, the system

automatically sends a reminder SMS text prompting the Marine to complete the requested task. When he does so, the system then automatically recalculates the risk score. This time the risk is 80% and the confidence acceptably narrow, so an alert is instantly generated.

Fig. 4 represents an abstract (due to space limitations) version of the rules which could be written to model the desired outcome for the use case. The first rule manages the results of the PTSD predictive model evaluation (see [18]), first checking that confidence in the output result is sufficient and then validating that the risk assessment itself is low enough for the patient not to require further evaluation. If either expectation is violated, appropriate actions are taken; either a new workflow to solicit additional information from the patient is started, and/or the patient is scheduled for additional testing. Notice that the same policies could have been written using standard rules, but the proposed syntax makes the definition more compact (1 rule instead of 4) and, most of all, ensures that the results of the constraint checks are recorded formally. Likewise, the second rule monitors the interaction between the patient and the provider, using the same pattern shown in section 4.1.

6 Conclusions

We have extended a production rule language to support the concept of expectations, the fulfillment or violation of which result in different consequences or behaviors. These syntactical additions enable a rule author to easily define and apply conformance criteria to the execution of the default rule logic. Rule expectations, if need be, can be cleanly deactivated and meta-reasoning disabled, thereby restoring more traditional rule behavior. The language extensions have been integrated into our reference run-time implementation, Drools, where it can exploits the engine's native complex event processing capabilities to better support temporal reasoning. Our next objective will be to research "fuzzy expectations" introducing uncertainty to the absolute fulfillment or violation of a constraint. Expectations that cope with the ambiguities of the real-world promise to provide important support for graded degrees of conformance testing.

Acknowledgements. We wish to thank the U.S. Navy KMR-II Project, the Health Sciences and Technologies - Interdepartmental Center for Industrial Research (HST-ICIR) - University of Bologna and the DEIS DEPICT Project which co-sponsored this research. The opinions of the authors do not necessarily state or reflect those of their respective employers, the United States Navy, the Department of Defense, or the United States Government, and shall not be used for advertising or product endorsement purposes.

References

1. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: The SCIFF framework. ACM Trans. Comput. Logic 9(4), 1–43 (2008)

2. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P., Sartor, G.: Mapping deontic operators to abductive expectations. CMOT 12(2-3), 205–225 (2006)
3. Allen, J.F.: Maintaining knowledge about temporal intervals. Commun. ACM 26(11), 832–843 (1983)
4. Damiani, G., Pinnarelli, L., Colosimo, S., Almiento, R., Sicuro, L., Galasso, R., Sommella, L., Ricciardi, W.: The effectiveness of computerized clinical guidelines in the process of care: a systematic review. BMC-HSR 10(1), 2 (2010), http://www.biomedcentral.com/1472-6963/10/2
5. Desai, N., Chopra, A.K., Singh, M.P.: Representing and reasoning about commitments in business processes. In: AAAI, pp. 1328–1333. AAAI Press (2007)
6. Forgy, C.: Rete: A fast algorithm for the many patterns/many objects match problem. Artif. Intell. 19(1), 17–37 (1982)
7. Fornara, N., Colombetti, M.: A commitment-based approach to agent communication. Applied Artificial Intelligence 18(9-10), 853–866 (2004)
8. Governatori, G.: Representing business contracts in *ruleml*. Int. J. Cooperative Inf. Syst. 14(2-3), 181–216 (2005)
9. Governatori, G., Hulstijn, J., Riveret, R., Rotolo, A.: Characterising Deadlines in Temporal Modal Defeasible Logic. In: Orgun, M.A., Thornton, J. (eds.) AI 2007. LNCS (LNAI), vol. 4830, pp. 486–496. Springer, Heidelberg (2007)
10. Governatori, G., Rotolo, A.: Norm Compliance in Business Process Modeling. In: Dean, M., Hall, J., Rotolo, A., Tabet, S. (eds.) RuleML 2010. LNCS, vol. 6403, pp. 194–209. Springer, Heidelberg (2010)
11. Governatori, G., Rotolo, A., Sartor, G.: Temporalised normative positions in defeasible logic. In: ICAIL, pp. 25–34. ACM (2005)
12. JBoss: JBoss Drools 5.2 - Business Logic Integration Platform (2011), www.jboss.org/drools
13. Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley Longman, Amsterdam (2002)
14. Montali, M., Pesic, M., van der Aalst, W.M.P., Chesani, F., Mello, P., Storari, S.: Declarative specification and verification of service choreographiess. TWEB 4(1) (2010)
15. Pesic, M., van der Aalst, W.M.P.: A declarative approach for flexible business processes management. In: Eder, J., Dustdar, S. (eds.) BPM Workshops 2006. LNCS, vol. 4103, pp. 169–180. Springer, Heidelberg (2006)
16. Sadiq, S.W., Governatori, G., Namiri, K.: Modeling control objectives for business process compliance. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 149–164. Springer, Heidelberg (2007)
17. Singh, M.P., Chopra, A.K., Desai, N.: Commitment-based service-oriented architecture. IEEE Computer 42(11), 72–79 (2009)
18. Sottara, D., Mello, P., Sartori, C., Fry, E.: Enhancing a production rule engine with predictive models using pmml. In: KDD 2011 (to appear, 2011)
19. Torroni, P., Chesani, F., Mello, P., Montali, M.: Social commitments in time: Satisfied or compensated. In: Baldoni, M., Bentahar, J., van Riemsdijk, M.B., Lloyd, J. (eds.) DALT 2009. LNCS, vol. 5948, pp. 228–243. Springer, Heidelberg (2010)
20. Torroni, P., Chesani, F., Yolum, P., Gavanelli, M., Singh, M.P., Lamma, E., Alberti, M., Mello, P.: Modelling Interactions via Commitments and Expectations. IGI Global (2009)
21. Yolum, P., Singh, M.P.: Flexible protocol specification and execution: applying event calculus planning using commitments. In: AAMAS, pp. 527–534. ACM (2002)