A model-driven approach for facilitating user-friendly design of complex event patterns

Juan Boubeta-Puig*, Guadalupe Ortiz, Inmaculada Medina-Bulo

Department of Computer Science and Engineering, University of Cádiz, C/ Chile 1, 11002 Cádiz, Spain

Abstract

Complex Event Processing (CEP) is an emerging technology which allows us to efficiently process and correlate huge amounts of data in order to discover relevant or critical situations of interest (complex events) for a specific domain. This technology requires domain experts to define complex event patterns, where the conditions to be detected are specified by means of event processing languages. However, these experts face the handicap of defining such patterns with editors which are not user-friendly enough. To solve this problem, a model-driven approach for facilitating user-friendly design of complex event patterns is proposed and developed in this paper. Besides, the proposal has been applied to different domains and several event processing languages have been compared. As a result, we can affirm that the presented approach is independent both of the domain where CEP technology has to be applied to and of the concrete event processing language required for defining event patterns.

Keywords:
complex event processing, model-driven development, event processing language, fast data

*Corresponding author. Tel.: +34 956 01 56 92
Email addresses: `juan.boubeta@uca.es` (Juan Boubeta-Puig), `guadalupe.ortiz@uca.es` (Guadalupe Ortiz), `inmaculada.medina@uca.es` (Inmaculada Medina-Bulo)

1. Introduction

In recent years, volumes of data produced by a variety of heterogeneous sources have increased around the world (Tsuchiya et al., 2012). As a result, both *Information Technology* (IT) and business users need to efficiently collect and process this huge amount of data in real time to discover relevant situations which will allow driving successful business decisions or actions (Hansen, 2013).

In this regard, *big data* is an approach which helps to process this huge amount of data. It is characterized in terms of the three *V's*: *Volume*, *Velocity* and *Variety* (Russom, 2011). Volume refers to the amount of data that can be managed and stored every day. Velocity is the big data dimension which deals with measuring how fast data can be collected and analyzed. Variety means the different existent data types: audio, video, text etc. However, big data normally focus on data previously collected and stored in databases. For that reason, it is not the best solution to process data from different sources in real time. To solve it, big data can be complemented with *fast data* (Hansen, 2013), an approach which allows to continuously analyze data and which can be characterized by a new dimension known as *Value*. This dimension aims to determine why such data is important for business.

In order to detect relevant or critical situations in business, fast data may be integrated with *Complex Event Processing* (CEP) (Luckham, 2002), a technology that allows detecting meaningful events in real time and inferring valuable knowledge for end users. For that purpose, the conditions describing the situations to be detected must be specified by using special templates known as *event patterns*. These patterns will be added into an event processing engine, the software responsible for analyzing and correlating the events received from different sources, as well as for raising alerts to users or systems interested in *complex events* (situations) generated by the detected event patterns.

These event patterns are defined using specific languages —developed for this purpose— known as *Event Processing Languages* (EPLs). Nevertheless, a wide experience on EPLs is required for defining such patterns. Thereby, one of the main drawbacks of using CEP by non-technical users, who are the ones having the domain-specific knowledge on the pattern to be detected, is the big learning curve necessary for becoming an expert in these languages. Some software solutions, such as Esper's editor (EsperTech Inc., 2013), Oracle CEP Visualizer (Oracle, 2013), StreamBase Studio (StreamBase, 2013) and SAP Sybase ESP Studio (Sybase, 2013), provide graphical tools to address this problem. Despite this fact, these tools are not user-friendly enough since non-experts on CEP have to write some EPL code by hand.

As a solution, in this paper, we propose a model-driven approach so that domain experts (but non experts on CEP) can concentrate in the graphical definition of event patterns in a user-friendly way without the need of hand-writing any code. Afterwards, the required code will be automatically generated. In concrete, our approach has four major contributions. Firstly, a metamodel is proposed to define event patterns as models which do not depend on the specific EPL required by the final engine used for complex event processing. Secondly, a domain-independent editor is implemented from this metamodel to facilitate user-friendly design of event patterns. Thirdly, a model validation process checks the correctness of these event patterns represented as models. Fourthly, a model transformation process automatically transforms such models into any particular EPL —Esper EPL code in this work. Furthermore, a case study in the field of health care is described and implemented for illustrating our proposal, which is also evaluated and discussed.

The rest of this paper is organized as follows. Section 2 includes background on *Model-Driven Software Development* (MDSD), CEP and EPL. Section 3 describes our model-driven approach in a nutshell and, afterwards, this approach is detailed in the following sections. In concrete, Section 4 explains the EPL metamodel for defining event patterns, Section 5 describes the implemented editor, Section 6 specifies the metamodel constraints and Section 7 details the process for transforming event pattern models into EPL code and the latter integration into a CEP engine. Then, Section 8 describes the application of our approach in a health-care case study. Subsequently, our approach is evaluated and discussed in Section 9. Some related works are described in Section 10. Finally, the conclusion and future work are highlighted in Section 11.

2. Background

In this section, the relevant subject matters for the scope of this paper, MDSD, CEP and EPL, are introduced.

2.1. Model-driven software development

MDSD is an important paradigm in software development which aims to find domain-specific abstractions and make them accessible by means of formal modeling (Stahl et al., 2006). These abstract representations of aspects of a system, known as *models*, are used as primary artifacts in the development process (Hussmann et al., 2011). The key features of this paradigm is that makes use of models of different levels of abstraction and provides *model transformations* in order to automatically transform a model into another as well as a model into implementation code.

Each model is an instance of a *metamodel*. In this scope a metamodel describes the structure of models in an abstract way. Particularly, a metamodel is defined using a metamodel language joined to a set of rules which specify the constraints so that the metamodel is well-formed. The most well-known metamodel language is Ecore and the *de facto* standard for capturing such constraints is *Object Constraint Language* (OCL).

This way, MDSD facilitates the automation of software production, increasing the productivity, quality and maintainability of software systems (Stahl et al., 2006). Even more, domain experts (non-technical users) can also understand such models, so that they can play an active role in software development.

2.2. Complex event processing

CEP is a cutting-edge technology which provides powerful techniques for processing and correlating events in order to detect relevant or critical business situations (complex events) in real time.

An event can be defined as anything that happens or could happen (Luckham, 2012). Mainly, events can be classified into three categories: a *simple event* is indivisible and happens at a point in time, a *complex event* contains more semantic meaning which summarizes a set of other events, and a *derived event* is generated when applying a process to one or more other events (Event Processing Technical and Society, 2011). Events can be derived from other events by applying or matching *event patterns*, templates where the conditions describing the situations to be detected are specified. A CEP engine is the software used to match these patterns over continuous and heterogenous event streams (timely ordered sequence of events of multiple types), and to raise alerts about the complex events created when detecting such event patterns.

According to Vincent (2010), *CEP systems*, as well as other decision-support systems such as *expert systems* take expert event-driven decisions, where expert knowledge is encoded from the available subject matter experts. In addition, these systems use "rules" (or event patterns) to determine whether stated goals (conditions) are fulfilled.

CEP can be applied to different areas. According to Luckham (2012, chap. 5), some of the major areas for sales of CEP are: fraud detection and security (Edge and Falcone Sampaio, 2012), transportation and traffic management (Dunkel et al., 2011), health care (Yuan and Lu, 2009; Yao et al., 2011), energy and manufacturing (Vikhorev et al., 2013), location-based services (Uhm et al., 2011), financial systems and operations (Edge and Falcone Sampaio, 2012), and operational intelligence in business (Chaudhuri et al., 2011). Among other additional areas, CEP can also be applied to home automation (Romero et al., 2011) and RFID signals (Yao et al., 2011).

To sum up, CEP allows detecting meaningful events and inferring valuable knowledge for end users in different domains. The main advantage of using this technology to process complex events is that the latter can be identified and reported in real time, reducing the latency in decision making, unlike the methods used in traditional software for event analysis.

2.3. Event processing language

As previously mentioned, in order to detect *situations of interests* on specific areas it is necessary the definition of so-called *event patterns*. These event patterns are defined using specific languages developed for this purpose known as EPLs. According to Etzion and Niblett (2010), these languages can be classified by the following language styles: stream-oriented, rule-oriented and imperative.

Stream-oriented EPLs are SQL-like languages but including new concepts, such as timing and temporal relationships. The learning curve is not high because their syntax is very close to SQL, worldwide known. Some of these EPLs are: Esper EPL (EsperTech Inc., 2013), CQL (Oracle, 2013), StreamSQL (StreamBase, 2013) and CCL (Sybase, 2013). In this work, we decided to transform graphical event patterns into Esper EPL since this language provides more operators than the others and its open-source engine is very efficient: it can process over 500,000 events/s (EsperTech Inc., 2013).

Rule-oriented EPLs implement event queries where condition expressions are evaluated over a set of facts. Some of CEP solutions that provide rule-oriented EPLs are: IBM Operational Decision Management (IBM, 2013), Drools Fusion (JBoss, 2013) and ETALIS (ETALIS, 2013).

Imperative EPLs define rules in an imperative way where operators define transformations over their inputs. Progress Apama (Progress Software, 2013) is an event processing platform which provides this EPL style.

Further information about other existing EPLs and CEP systems can be found in the survey by Cugola and Margara (2012).

3. Our approach in a nutshell

This section outlines the main contributions of our model-driven approach to facilitate user-friendly design of complex event patterns. The ultimate goal of this approach is the creation of a domain-independent editor allowing non-technical users to graphically define event patterns in a user-friendly way without the need of hand-writing any code. To reach this goal, we have followed the steps below:

Firstly, a metamodel to enable models (event patterns) definition through the use of such an editor has been defined (Section 4). One of the key aspects of this metamodel is that event pattern definition does not depend on the specific EPL required by the engine used for complex event processing. Therefore, every event pattern is graphically designed by the user once and then it can be automatically transformed into any particular EPL, such as Esper EPL, CQL, StreamSQL or CCL, among others.

Secondly, a graphical editor has been implemented from this metamodel (Section 5). This editor makes possible non-technical users to concentrate in the definition of the relevant or critical situations to be detected based on their expertise knowledge. These definitions can be reused on different IT systems requiring the implementation of these event patterns in a different specific language. According to Luckham (2012), "*we must not confuse what the event pattern is with how we specify it using a language*".

Thirdly, a model validation process which checks the correctness of these event patterns represented as models has been provided (Section 6). Particularly, checking the correctness of a model includes examining three kinds of properties: consistency —a model is inconsistent if it has some contradictions, completeness —a model is incomplete if there are missing elements which should be used to give an adequate specification— and validation —it checks if the model formalizes the requirements correctly.

Finally, Section 7 defines a model transformation process in order to automatically transform such models into any particular EPL. In addition, the generated code will be then automatically inserted in a concrete CEP engine. Thanks to the transformations rules defined in this work, such patterns can be automatically transformed into Esper EPL code and inserted in an Esper engine.

4. The EPL metamodel

This section describes our metamodel for describing event patterns in a user-friendly way. Defining these patterns as models allows domain experts to be oblivious to the complexity of the concrete syntax corresponding to the specific EPL which will be used to implement them.

Figure 1 shows the main metaclasses in our metamodel and their relationships. We have only represented the main metaclasses to facilitate the comprenhension of the metamodel. The remaining metaclasses are described in the following tables: a detailed description of the types of *Operator* metaclass (*ConditionOperator*, *PatternOperator* and *OutOperator*) is given in Table 1, a detailed description of the types of *Operand* metaclass (*ConditionOperand*, *PatternOperand* and *OutOperand*) is given in Table 2 and a description about *DataWindow* and *PatternTimer* is given in Table 3. The main metaclasses in Figure 1 are described below following top-down and left-right order:

EPLModel It is the main concept of the metamodel and represents EPL sentences. EPL models consist of three kinds of components: *search conditions*, *pattern* and *output*. In order to establish the relationships between the elements contained in these components, *links* are used.

Link It defines the graphical representation of one or more relationships between *operators* and *operands*. Each *link* is characterized by an *order* $(0, 1 \ldots N)$ that is used to enumerate the *operands* that participate in a relationship; *order* set to 0 indicates that it is not relevant in that relationship.

Operator It is used to express a specific operation between one or more *operands*. The description of the types of *Operator* metaclass is described in Table 1.

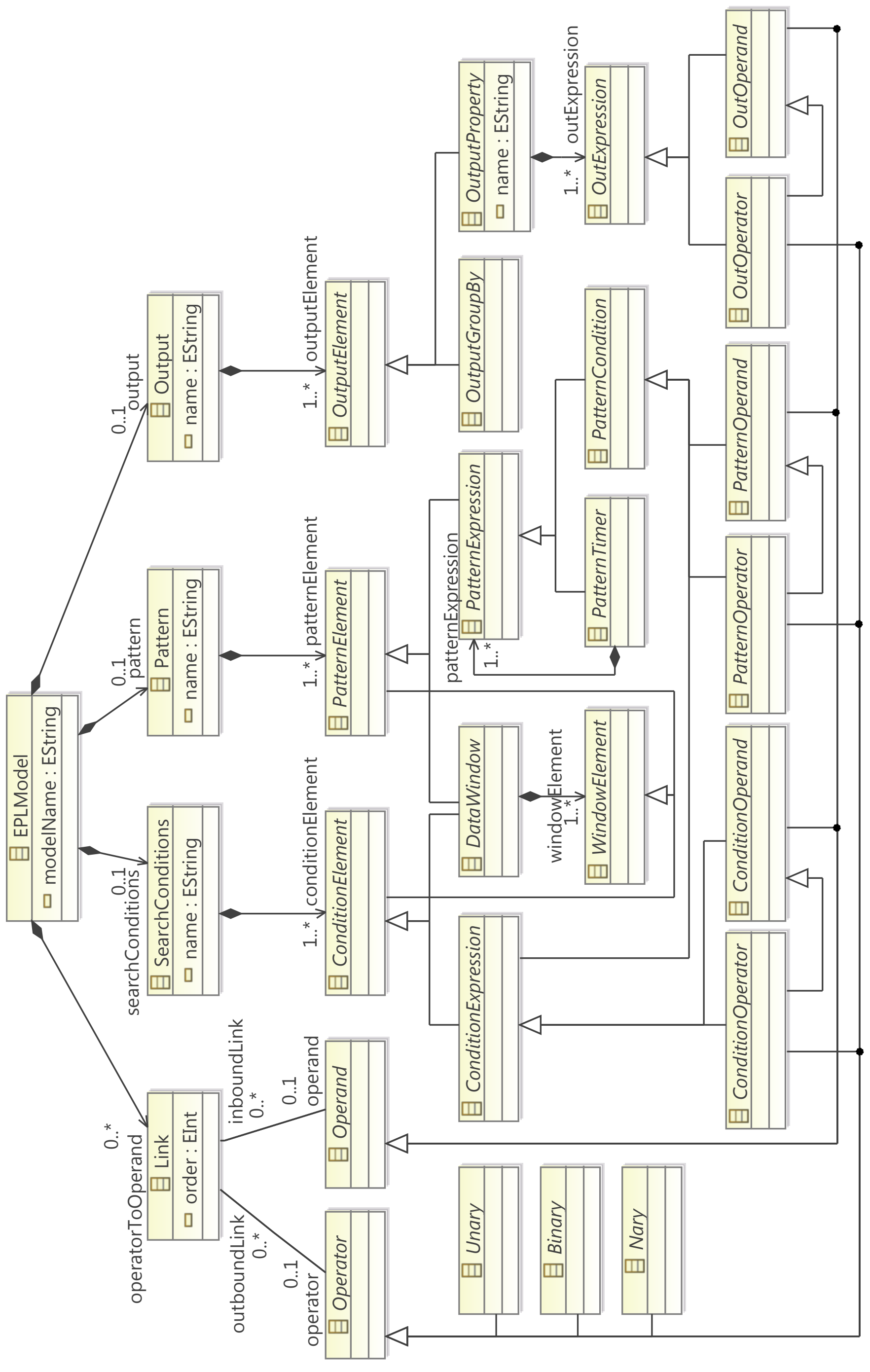


Figure 1: EPL metamodel.

Type	**Subtype**	**Operator**	**Arity**	**Description**
PatternOperator		*Every*	U	It keeps firing when encountering the specified *pattern expression.*
		EveryDistinct	U	It is similar to *Every*, but eliminating duplicate results according to a given *distinct-value* expression.
		FollowedBy	N	It determines a *pattern expression* must be *followed by* another.
		Range	U	It specifies the minimum number of times which the *pattern expression* associated with *Until* must become true.
		Repeat	U	It defines how many times a *pattern expression* must be repeated.
		Until	B	It matches a *pattern expression until* the condition (another *pattern expression*) is evaluated to true.
		While	B	It matches a *pattern expression while* the condition (another *pattern expression*) is evaluated to true.
PatternOperator & *ConditionOperator*	*Logical*	*And*	N	It returns a true *value* only if all *operands* are true.
		Not	U	It returns a true *value* if the *operand* is false, and a false *value* if the *operand* is true.
		Or	N	It returns a true *value* if at least one *operand* is true.
ConditionOperator	*Comparison*	*Equal*	B	It returns a true *value* if $operand1 = operand2$.
		GreaterEqual	B	It returns a true *value* if $operand1 \geq operand2$.
		GreaterThan	B	It returns a true *value* if $operand1 > operand2$.
		LessEqual	B	It returns a true *value* if $operand1 \leq operand2$.
		LessThan	B	It returns a true *value* if $operand1 < operand2$.
		NotEqual	B	It returns a true *value* if $operand1 \neq operand2$.
ConditionOperator & *OutOperator*	*Arithmetic*	*Addition*	B	It sums two numeric *values.*
		Division	B	It divides one numeric *value* by another.
		Modulus	B	It returns the remainder of dividing one numeric *value* by another.
		Multiplication	B	It multiplies two numeric *values.*
		Subtraction	B	It subtracts one numeric *value* from another.
OutOperator	*Aggregate Function*	*Avg*	U	It returns the average of the *values* in an expression.
		Count	U	It returns the number of the *values* in an expression.
		Max	U	It returns the highest *value* in an expression.
		Min	U	It returns the lowest *value* in an expression.
		Sum	U	It adds the *values* in an expression.

Table 1: Metamodel operators (U = Unary, B = Binary, N = N-ary).

Type	**Operand**	**Description**
PatternOperand	*AtTimer*	It turns into true at a defined time (*minutes*, *hours*, *daysOfMonth*, *months*, *daysOfWeek* or *seconds*).
	IntervalTimer	It waits for the specified *time period* (*years*, *months*, *weeks*, *days*, *hours*, *minutes*, *seconds* and *milliseconds*) before turning to true.
PatternOperand & *ConditionOperand*	*Event*	An *event* is a domain relevant concept. Every *event* must have a *type name* and can contain different *properties*.
ConditionOperand	*Property*	A *property* describes a feature of an *event*. Every *property* must have a *name* associated with one of the following types: *boolean*, *double*, *float*, *integer*, *long* or *string*. A *property* can be linked to different *property references*.
ConditionOperand & *OutOperand*	*Value*	It defines a *boolean*, *double*, *float*, *integer*, *long* or *string* value.
OutOperand	*PropertyReference*	It allows to reference a *property*. Therefore, a *property* is created once into an *event* and can be used several times by means of this *operand*.

Table 2: Metamodel operands.

Type	**Window/Timer**	**Description**
DataWindow	*Length*	Sliding window by the specified number of *events* (*size*).
	LengthBatch	Tumbling window up to the specified number of *events* (*size*).
	Time	Sliding window by the specified *time period* (*years*, *months*, *weeks*, *days*, *hours*, *minutes*, *seconds* and *milliseconds*).
	TimeAccumulating	Sliding window accumulating *events* until no more *events* arrive within a given *time period.*
	TimeBatch	Tumbling window that batches events and releases them every specified *time period.*
	TimeLengthBatch	It is a combination of *LengthBatch* and *TimeBatch* windows.
PatternTimer	*WithinTimer*	It is permanently evaluated to false if the associated *pattern expression* does not turn to true during the specified *time period.*
	WithinMaxTimer	It is similar to *WithinTimer.* It is permanently evaluated to false if the associated *pattern expression* does not turn to true during the specified *time period*, or the number of matches reaches the *maxCount* counter.

Table 3: Metamodel data windows and pattern timers.

Operand A data on which an *operator* is performed. The description of the types of *Operand* metaclass is described in Table 2.

Unary A unary *operator* has only an *operand*. Therefore, it is not necessary to specify the *order* of the *link* that connects the *operator* with the *operand*.

Binary A binary *operator* has two *operands*. Some binary *operators* require the definition of *order* for the *links* connected to their *operands*.

Nary A n-ary *operator* has two or more *operands*. The *Followed-By* operator requires the definition of *order* for the *links* connected to their *operands*.

SearchConditions This component represents the conditions that can be defined to join event streams or filter events. These conditions can be defined by means of *data windows* and *condition expressions*.

DataWindow It specifies a bounded set of events from an event stream. A *data window* can contain other *data windows*, *condition expressions* or *pattern expressions*. The descriptions of *data windows* are given in Table 3.

ConditionOperator It defines the types of *operators* which can be contained into a *search conditions* component: *arithmetic*, *comparison* and *logical*.

ConditionOperand It defines the types of *operands* which can be contained into a *search conditions* component: *value*, *property* and *event*.

Pattern An event *pattern* is a template specifying conditions which can match sets of related events. These conditions can be defined by means of *data windows* and *pattern expressions*. A *pattern condition* can be described by means of *pattern operands*, *pattern operators* and *condition expressions*.

PatternTimer It represents the time conditions for an event *pattern*. A *pattern timer* can contain other *pattern timers* or *pattern conditions*. The descriptions of *pattern timers* are given in Table 3.

PatternOperator It describes the types of *operators* which can be contained into a *pattern* component: *logical* and specific *pattern operators*.

PatternOperand It describes the types of *operands* which can be contained into a *pattern* component: *event* and *pattern timer*.

Output This component defines the results to be obtained from the *search condition* or *pattern* components. These results can be summarized as a set of other events, in this case creating a complex event.

OutputGroupBy It divides the *output* into groups of one or more event property names. It contains one or more *property references* linked to such event *properties*.

OutputProperty It describes one of the *properties* which form the EPL *output*. It contains one or more *out expressions*.

OutOperator It represents the types of *operators* which can be contained into an *output property*: *aggregate function* and *arithmetic*.

OutOperand It represents the types of *operands* which can be contained into an *output property*: *value* and *property reference*.

5. The event pattern editor

The Epsilon family of languages (Kolovos et al., 2013) has been used to implement this editor. Epsilon provides EuGENia (Epsilon, 2013), a tool which generates automatically the models needed for the implementation of a *Graphical Modeling Framework* (GMF) editor resulting from a single annotated Ecore metamodel. This editor allows to create models (event patterns) conforming to our EPL metamodel which will be saved as *XML Metadata Interchange* (XMI) files.

There are two relevant elements in the editor: the palette and the canvas.

The palette has been customized using *Epsilon Object Language* (EOL) (Kolovos et al., 2013, chap. 3), an imperative language inspired by OCL, classifying its elements into the following ten groups:

Connections It contains the *Link* tool for linking *operators* to *operands*. It also contains the *PropertyToReference* tool for linking event properties (*Property*) to references of event properties (*PropertyReference*).

Components It groups the tools corresponding to the following metamodel classes: *Output*, *Pattern* and *SearchConditions*.

Elements It groups the tools corresponding to the following metamodel classes: *Event*, *OutputProperty*, *OutputGroupBy*, *Property*, *PropertyReference* and *Value*.

Arithmetic Operators It contains tools for all *arithmetic operators* defined in our metamodel.

Comparison Operators It contains tools for all *comparison operators* defined in our metamodel.

Logical Operators It groups the tools for all *logical operators* defined in our metamodel.

Pattern Operators It contains all *pattern operators* defined in our metamodel, except *And*, *Or* and *Not* operators which have been already included in the *Logical Operators* group.

Pattern Timers It includes both pattern operands (*AtTimer* and *IntervalTimer*) and pattern timers (*WithinTimer* and *WithinMaxTimer*).

Aggregate Functions It specifies tools for the *aggregate functions* included in our metamodel: *Avg*, *Count*, *Max*, *Min* and *Sum*.

Data Windows It contains the tools for *data windows* described in our metamodel.

The canvas is the editor area where the elements in the palette can be inserted, in a drag-and-drop fashion, to define models that conform to our metamodel. These elements' attributes can be set both in a graphical way and using the application's properties view.

This canvas can only contain *components*, i.d. domain experts can only drag and drop *SearchConditions*, *Pattern* or *Output* components into the canvas. The rest of palette tools can be drag and drop into such components. Notice that a particular palette tool can be used in a specific component if the obtained model is conformed to our metamodel and is also correct according to the metamodel constraints defined in Section 6. Some screenshots which illustrate the implemented graphical modeling editor can be seen in Section 8.2.

6. Model validation

This section describes the model validation process which checks the correctness of event patterns represented as models (metamodel instances). To this end, we enriched our metamodel with the following textual constraints for specific metaclasses:

- It must be named: *EPLModel*, *Event* and *Property* metaclasses.
- *Pattern* or *SearchConditions* must be included: *EPLModel* metaclass.
- An *operator* cannot be linked to itself or another identical *operator*: *Link* metaclass.
- It must be linked to *pattern operands*: *PatternOperator* metaclass.
- It must be linked to *condition operands*: *ConditionOperator* metaclass.
- It must be linked to *out operands*: *OutOperator* metaclass.
- It must have 1 outbound *link*: *Unary* metaclass.
- It must have 2 outbound *links* with *orders* 1 and 2, or both 0: *Binary* metaclass.
- It must have at least 2 outbound *links* with *orders* $1, 2 \ldots N$, or all set to 0 (except *FollowedBy*): *Nary* metaclass.
- It cannot be linked to *Event* and both *operands* must have the same data type: *Arithmetic* and *Comparison.*
- An appropiate *value* must be set according to its data type: *Value* metaclass.
- It must contain unique *properties*: *Event* metaclass.
- All contained *PropertyReference* must be unique: *EveryDistinct* and *OutputGroupBy* metaclasses.
- *lowEndpoint* $\leq$ *highEndpoint*: *Range* metaclass.
- None of *referenced properties* must be used with *aggregate functions*: *OutputGroupBy* metaclass.

- Attributes must be positive: *AtTimer*, *EveryDistinct*, *IntervalTimer*, *Length*, *LengthBatch*, *LengthBatchWithinTimer*, *Repeat*, *Time*, *TimeAccumulating*, *TimeBatch*, *TimeLengthBatch*, *WithinTimer* and *WithinMaxTimer* metaclasses.

These metamodel constraints have been defined using *Epsilon Validation Language* (EVL) (Kolovos et al., 2013, chap. 4). Although OCL is the *de facto* standard for capturing constraints in modeling languages, it has several shortcomings compared to EVL. OCL does not support meaningful messages that can be reported to the end user. In addition, OCL supports only errors (not warnings) and does not accept repairing inconsistencies as EVL does, among others.

7. EPL code generation and insertion in a CEP engine

Once event patterns have been modeled by domain experts and automatically validated with the help of our implemented editor, the last step of our model-driven approch consists of transforming the event pattern models into EPL code.

As previously mentioned, one of the relevant features of our approach is that domain experts only have to define event patterns once. Afterwards, our editor will be able to transform them into different EPLs. To this end, it will be necessary to create a new module which allows us to transform the defined patterns into a particular EPL.

In order to transform these graphical models into Esper EPL code, we have created a module using *Epsilon Generation Language* (EGL) (Kolovos et al., 2013, chap. 7), a language for model-to-text transformation. In addition, this module will automatically deploy this EPL code into Esper engine. For this purpose we have made use of the API provided by this engine.

It is important to highlight that, although we have only transformed event pattern models into Esper EPL, our editor can be extended to support other EPLs, creating such a module per EPL to be incorporated.

8. Defining event patterns with the editor

In this section, we specify the phases to be followed to make use of our approach for facilitating user-friendly design of complex event patterns . Afterwards, we apply it to a health-care scenario. In concrete, we propose and

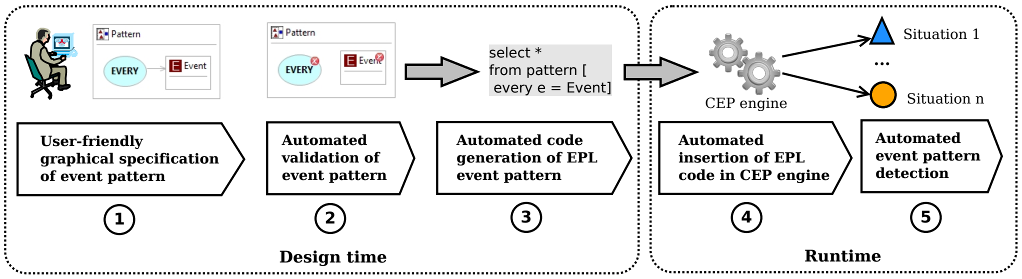

Pattern
EVERY
Event
Pattern
EVERY
Event
select *
from pattern [
every e = Event]
CEP engine
Situation 1
...
Situation n
User-friendly graphical specification of event pattern
Automated validation of event pattern
Automated code generation of EPL event pattern
Automated insertion of EPL code in CEP engine
Automated event pattern detection
1
2
3
4
5
Design time
Runtime

Figure 2: Phases of our approach for defining event patterns in a user-friendly way.

define some event patterns to detect influenza outbreaks around the world. Then, these patterns are defined graphically making use of our editor.

8.1. Overview

The phases to be followed for defining event patterns with our model-driven approach are detailed below (see Figure 2):

1. **Event pattern modeling**: the domain expert is responsible for graphically defining the event patterns to be detected in a specific scenario, such as health care or stock market. The obtained models (event patterns) will conform to our metamodel described in Section 4.
2. **Event pattern validation**: once an event pattern is modeled, the editor will validate it by means of the metamodel constraints defined in Section 6. If the model is not correct, then the editor will show the errors which must be solved before going on.
3. **Model-to-code transformation**: the event pattern model will be automatically transformed into EPL code. This code will depend on the specific EPL provided by the chosen CEP engine, as explained in Section 7.
4. **Insertion of EPL code in a CEP engine**: the EPL code of the modeled event pattern will be automatically inserted into the CEP engine at runtime (also explained in Section 7).
5. **Pattern detection**: from this moment, the engine will be able to detect the new critical or relevant business situation described by the EPL event pattern recently added to the engine.

Notice that the first three phases are executed at design time and the last ones at runtime.

8.2. Case study of influenza outbreaks

In this case study we illustrate our approach in the domain of health care. In particular, we propose to detect epidemics and pandemics of influenza in real time around the world. Thereby, CEP would allow health officials to mitigate as soon as possible the impact of epidemics and global pandemics.

8.2.1. Event patterns for detecting influenza outbreaks

In order to detect influenza outbreaks by means of CEP technology, it is necessary the definition of a set of event patterns for this domain. To do this, we did some research in this field, thanks the help of specialist health books and websites (WHO, 2013; Longo et al., 2012) as well as the expertise and knowledge of Spanish National Health System's workers.

According to real requirements for detecting influenza cases, we defined the following complex event patterns:

Influenza possible case This pattern detects possible occurrences of influenza cases, when the following conditions are fulfilled:

1. The patient has fever (above 38 °C) and myalgia.
2. The patient has mucus, sneeze, sore throat or cough.
3. The patient has chill, headache or fatigue.

Influenza epidemic case There are 25 influenza possible cases of influenza in a particular country during 5 days.

Influenza pandemic case There are 2 or more influenza epidemic cases during 3 days.

Notice that the number of cases and days for detecting epidemic and pandemic cases would be different depending on external factors from different countries. Nevertheless, this fact is not relevant to demonstrate the usability and functionality of our editor since the domain expert can graphically change such numbers at any moment.

The following step consists of graphically defining the event patterns using our implemented editor.

8.2.2. Graphical definition and EPL code of influenza possible case

Figure 3 highlights the model of *influenza possible case* pattern obtained as a result of using the different tools provided by the editor palette.

First of all, a *Pattern* component is dropped into the editor canvas in order to define the conditions to be matched. An *event* element is then included into *Pattern* and named as *PatientStatus*, the simplest event in this scenario which represents a particular status of a patient which visits a doctor in a hospital. All events are represented by *E* letter in the model. Its properties, represented by *P* letter, are the following: the *timestamp* when the patient status was registered in the system, the *id* of this patient, the *location* where the patient was reviewed by the doctor, and the symptoms which the patient had: *temperature*, *myalgia*, *mucus*, *sneeze*, *sore throat*, *cough*, *chill*, *headache* and *fatigue*. Afterwards, such symptoms are connected by comparison and logical operators to define the conditions of the pattern. Besides, the *every* pattern operator is connected to *PatientStatus* in order to analyze all events of this type.

If the conditions of the pattern are matched, then we are interested in creating a new event of *InfluenzaPossibleCase* type, which will summarize such situation. This complex event will only have three new properties: *registrationTime*, *patientId* and *possibleCaseLocation* that are the same properties of *PatientStatus* (*timestamp*, *id* and *location*, respectively) but with other more meaningful names, i.d. these new properties are alias of the other ones. These associations are made by means of *PropertyToReference* links.

The model is conformed to our metamodel, and therefore it is properly validated. We are aware of this because there are no cross symbols in the figure indicating that there are problems with any element in the model. An example of this type of errors can be seen in Section 8.2.4.

Once the model is validated, it can be automatically transformed into EPL code. To achieve this goal, the domain expert will select the option *Transform the model into EPL code* of the context menu of the editor. The EPL code generated by the editor is shown below:

```
@Name('InfluenzaPossibleCase')
insert into InfluenzaPossibleCase
select p.timestamp as registrationTime, p.id as patientId,
   p.location as possibleCaseLocation
from pattern[every p = PatientStatus(
   (temperature > 38) and myalgia
```

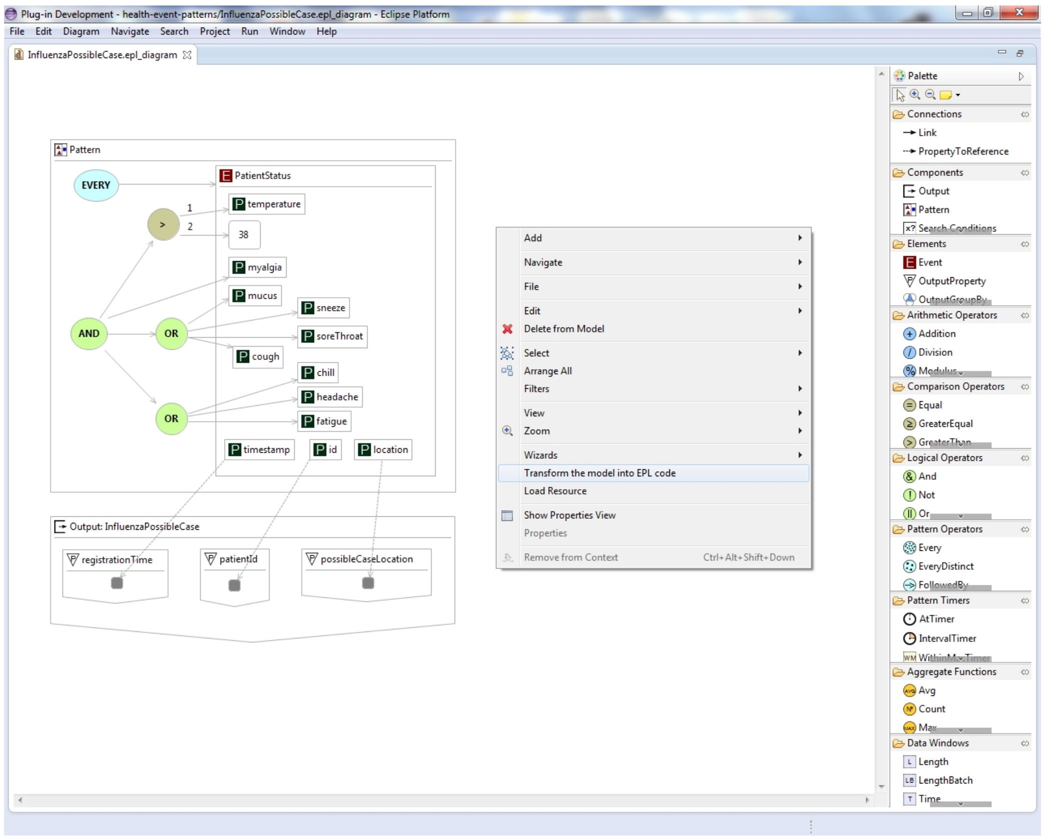


Figure 3: Event pattern model of influenza possible case.

```
   and (mucus  or sneeze or soreThroat or cough)
   and (chill or headache or fatigue))]
```

Notice that the `select` clause specifies the event *properties* or *events* to be retrieved, and the `from` clause indicates the pattern (or event streams) to be used. The `insert into` is used to make available the results of the event pattern as an event stream (complex events) so that they can be used in further event patterns —it can also be used to merge multiple event streams in order to form a single event stream.

Finally, according to our approach, this code will be dynamic and automatically inserted into the CEP engine to start detecting this new pattern.

8.2.3. Graphical definition and EPL code of influenza epidemic case

Figure 4 shows the graphical definition of influenza epidemic case. In this case, the conditions must be satisfied within 5 days. For that purpose, all the conditions are included into a 5-day *WithinTimer*.

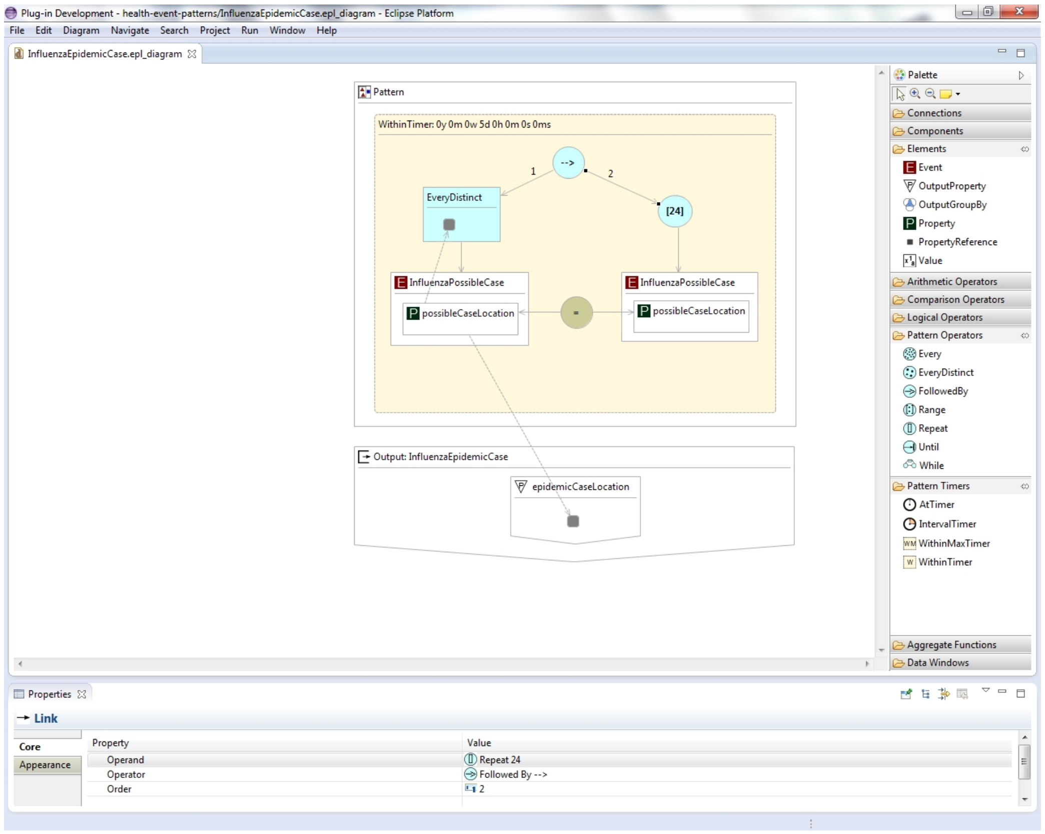
As previously mentioned, this pattern is detected if there are 25 influenza possible cases of influenza in a particular country during 5 days. In other words, it is satisfied if there is an *InfluenzaPossibleCase* event followed by 24 more *InfluenzaPossibleCase*, with the restriction that all of these 25 events belong to the same country (*location*). Because of we are not interested in detecting more than one influenza epidemic by country, in this model we use the *every-distinct* operator containing the reference to the *location* property, instead of the *every* operator.

As represented by the *output* component, if the pattern is satisfied then *InfluenzaEpidemicCase* complex events will be created with a new property: *epidemicCaseLocation*.

In Figure 4 it can be checked that end users will be able to define the value of properties graphically or by making use of the property panel.

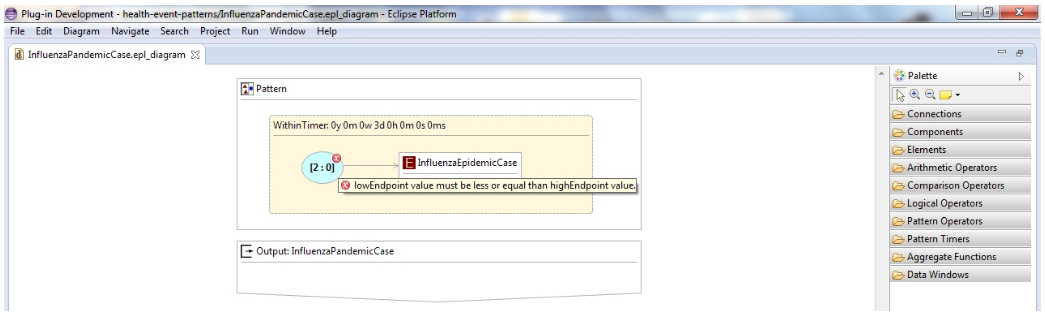
The EPL code generated by the editor is shown below:

```
@Name('InfluenzaEpidemicCase')
insert into InfluenzaEpidemicCase
select i.possibleCaseLocation as epidemicCaseLocation
from pattern[every-distinct(i.possibleCaseLocation)
   i = InfluenzaPossibleCase -> [24] InfluenzaPossibleCase(
   possibleCaseLocation = i.possibleCaseLocation)
   where timer:within(5 days)]
```



Property	Value
Operand	Repeat 24
Operator	Followed By -->
Order	2

Figure 4: Event pattern model of influenza epidemic case.


Plug-in Development - health-event-patterns/InfluenzaPandemicCase.epl_diagram - Eclipse Platform
File Edit Diagram Navigate Search Project Run Window Help
InfluenzaPandemicCase.epl_diagram
Pattern
WithinTimer: 0y 0m 0w 3d 0h 0m 0s 0ms
[2 : 0]
InfluenzaEpidemicCase
Output: InfluenzaPandemicCase
Palette
Connections
Components
Elements
Arithmetic Operators
Comparison Operators
Logical Operators
Pattern Operators
Pattern Timers
Aggregate Functions
Data Windows

lowEndpoint value must be less or equal than highEndpoint value.

Figure 5: Event pattern model of influenza pandemic case.

8.2.4. Graphical definition and EPL code of influenza pandemic case

Figure 5 sketches the graphical definition of influenza pandemic case. In this case, a 3-day *WithinTimer* contains an *InfluenzaEpidemicCase* event, which is linked by the *range* repetition operator.

As the figure shows, this model does not conform to the metamodel since the defined range operator is not correct. The error message reports that "*lowEndpoint value must be less or equal than highEndpoint value*". So this model would not be transformed into EPL code as long as such operator is properly defined.

In this model, the *output* component —called *InfluenzaPandemicCase*— is empty. This means that the complex events generated by this event pattern will have the same properties of *influenza-epidemic-case* event type.

The EPL code generated by the editor is shown below:

```
@Name('InfluenzaPandemicCase')
insert into InfluenzaPandemicCase
select *
from pattern[[2:242] InfluenzaEpidemicCase
   where timer:within(3 days)]
```

Notice that this code has been generated by the editor once the end user has changed the wrong value of *highEndpoint* into 242, which is greater than 2.

9. Evaluation and discussion

In this section we demonstrate that our model-driven approach for defining event patterns allows us to transform them into different EPLs. Therefore, it makes possible that the event patterns designed by domain experts can be reused in IT systems that use different CEP engines to generate relevant alarms in real time. Furthermore, we check that our approach can be applied to different domains where CEP technology is required.

9.1. Metamodel evaluation

We have evaluated that our EPL metamodel allows us to define event patterns as models regardless of the EPL used by a concrete CEP engine. For this purpose, it is fundamental to check if the event pattern models, conformed to our metamodel and created with our editor, can be transformed into other EPLs apart from Esper EPL. As previously mentioned, we decided to generate Esper EPL code in this work since it provides more pattern operators compared to the other evaluated languages.

To demonstrate such assertion, we have done a comparison (see Table 4) where we determine how every metaclass of the model elements used in any of the event patterns graphically defined in this paper would be transformed into Esper EPL, CQL, StreamSQL and CCL code, among others. These metaclasses have been selected as representative samples for the paper, but we conducted this study for all the metaclasses in the model.

We summarizes the most important aspects of this comparison in the following paragraphs:

Table 4 shows how some metaclasses are equivalent to more than one clause in EPL code. Concretely, *SearchConditions* component is equivalent to both `from` and `where` clauses while *Output* component is equivalent to both `insert into` and `select`. We have defined our metamodel in that way so that domain experts do not have to know what these specific clauses mean; they only have to focus on designing the *search conditions* and the *output* they would like to obtain.

The *Every* metaclass is not equivalent to any StreamSQL or CCL operator. This is because when an StreamSQL or CCL *pattern* is defined it will automatically look for all events which can be matched, without the need of using a specific operator, but providing the same functionality that *Every* operator.

The *EveryDistinct* metaclass is not equivalent to any CQL or CCL operator. However, there is an alternative to *EveryDistinct* which consists of using *Every* plus *And* and *Not* operators. For instance, the Esper expression ***`every-distinct`***`(a.id) a = Event` is equivalent to ***`every`*** `a = Event` ***`and not`*** `b = Event(b.id = a.id)`. Taking this equivalence into account,

Metaclass	**Esper EPL**	**CQL**	**StreamSQL**	**CCL**
SearchConditions	`from`	`FROM`	`FROM`	`FROM`
	`where`	`WHERE`	`WHERE`	`WHERE`
Pattern	`from pattern`	`MATCHING`	`MATCHING`	`FROM PATTERN`
Output	`insert into`	`INSERT INTO`	`INSERT INTO`	`INTO`
	`select`	`SELECT`	`SELECT`	`SELECT`
	`as`	`AS`	`AS`	`AS`
OutputGroupBy	`group by`	`GROUP BY`	`GROUP BY`	`GROUP BY`
Every	`every`	`EVERY`	By default	By default
EveryDistinct	`every-distinct`		`ONCE`	
FollowedBy	`->`	`FOLLOWED BY`	`,`	`->` and `THEN`
Range	`[a:b]`	`BETWEEN a AND b`	`BETWEEN a AND b`	`BETWEEN a AND b`
Repeat	`[n]`	`BETWEEN n AND n`	`BETWEEN n AND n`	`BETWEEN n AND n`
And	`and` and `,`	`AND`	`&&` and `AND`	`&&` and `AND`
Or	`or`	`OR`	`||` and `OR`	`||` and `OR`
Equal	=	=	=	==
GreaterThan	>	>	>	>
Subtraction	−	−	−	−
Time	`win:time (n seconds)`	`RETAIN n SECONDS`	`KEEP n SECONDS`	`SIZE n`
WithinTimer	`timer:within (n seconds)`	`WITHIN n SECONDS`	`n SECONDS:`	`WITHIN n TIME`

Table 4: A comparison between the metaclasses of model elements used in any of the event patterns graphically defined in this paper and their equivalent Esper EPL, CQL, StreamSQL and CCL code.

EveryDistinct metaclass can be transformed into CQL or CCL code making use of *Every* operator.

The *Range* and *Repeat* metaclasses can be transformed into the equivalent operator `BETWEEN ... AND ...` —this operator is also provided by Esper EPL— since CQL, StreamSQL and CCL do not distinguish between *range* and *repeat* operators.

The *And* and *Or* metaclasses can be transformed into different StreamSQL and CCL operators depending on if they are used as logical operators or pattern operators. In the case of logical operators, the *And* and *Or* metaclasses are equivalent to `&&` and `||`, respectively. In the case of pattern operators, these metaclasses are equivalent to `AND` and `OR`, respectively.

Therefore, from this comparison it can be concluded that all model elements used in the graphical definition of event patterns in this paper can also be transformed into CQL, StreamSQL and CCL code, among others —not only for Esper. To that end, we would have to create a new model-to-text module in our editor for each new language to be used (see Section 7). This fact emphasizes the usefulness of our approach, making possible that every event pattern is graphically defined by expert domains once, and automatically transformed into different EPL codes, should it be necessary.

9.2. Editor and EPL code generation evaluation

We previously showed how our model-driven approach was applied to a health-care domain where we created new health event patterns from scratch (see Section 8).

In order to prove that our approach can be applied to different domains —and not only to health domain— we illustrate below how other existing event patterns, defined by other authors in a different domain, can also be designed and implemented with our editor.

Particularly, the event patterns described by Dunkel et al. (2011) to detect relevant situations in the domain of road traffic management have been modeled by means of our editor.

Because of space limitations on this paper, we only illustrate one of such event patterns. Concretely, the event pattern which notifies about retention caused by a trunk incident. The EPL code of this pattern is as follows:

```
insert into Problem
select Aft.id as location,
   'retention because of trunk incident' as description,
   'linear connection' as type,
   Aft.demand - Aft.capacity as excess,
   'incident' as state,
   'problem' as category
from Section.win:time (30 seconds) Aft,
   Section.win:time (30 seconds) Bef
where Bef.next_section = Aft.id
   and (Bef.speed = 'LOW' or Bef.occupancy = 'HIGH')
   and (Aft.density = 'LOW' or Aft.density = 'MEDIUM')
   and (Aft.occupancy = 'LOW' or Aft.speed = 'HIGH')
group by Aft.id
output last every 30 seconds;
```

Notice that the `where` clause describes *search conditions* that specify which *event* or event combination should be detected. According to these authors, "*this pattern matches when a section is characterized by low speed or high occupancy and, in the subsequent section, shows low or medium density, and either high speed or low occupancy*".

Figure 6 sketches the graphical definition of this pattern using our editor. As shown in the figure, the pattern model obtained is understandable and user-friendly.

It is important to highlight that the code generated by our editor is the same as the code shown in Dunkel et al. (2011). Therefore, we can affirm that our editor can adequately model event patterns in different domains and can correctly generate the EPL code for these patterns.

10. Related work

Nowadays there are several proprietary and academic editors for defining event patterns and transforming them into their own EPL code. Although most of these editors provide a metamodel implementation, they are not user-friendly enough since they still require that non-experts on CEP have to write some EPL code by hand. In the following paragraphs, we are going to examine the most representative approaches for each type of EPL.

Taking into account stream-oriented EPLs, Esper Enterprise Edition from EsperTech offers an editor based on Adobe Flash/Flex technology (EsperTech Inc., 2013). Notice that although the Esper engine is open source, this editor is not. Another editor (SocEDA, 2013) has been developed for EPL

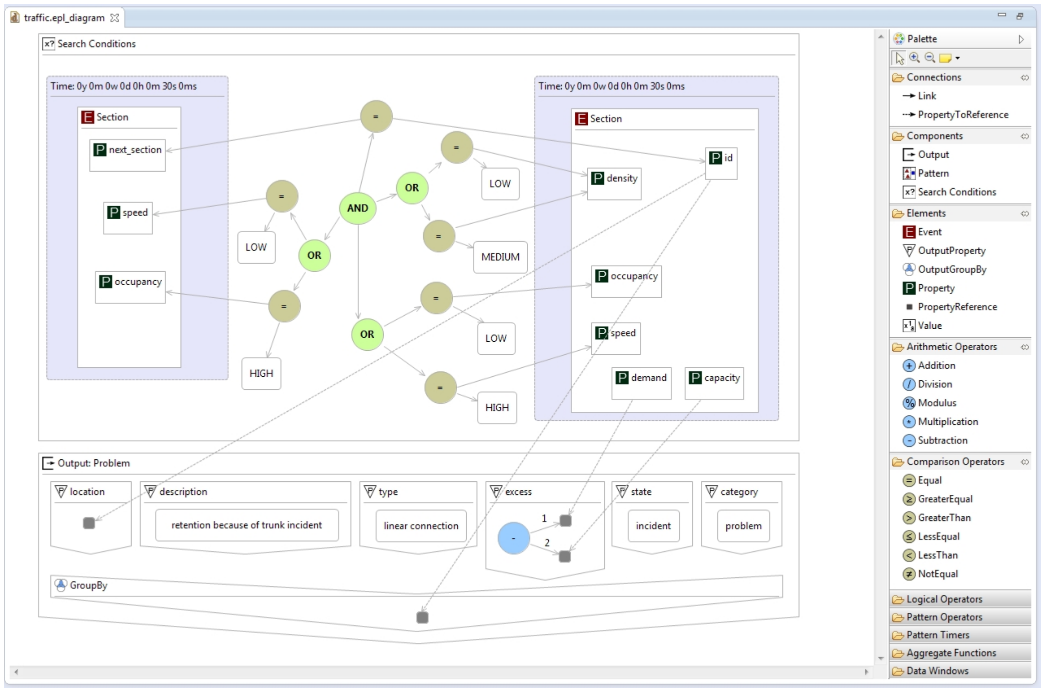


Figure 6: Event pattern model of traffic problem.

Esper in SocEDA (SOCial Event Driven Architecture), an ANR (National Research Agency) project. Furthermore, Oracle provides the Oracle CEP Visualizer (Oracle, 2013), StreamBase offers StreamBase Studio (StreamBase, 2013), and Sybase develops SAP Sybase ESP Studio (Sybase, 2013). Even though these editors allow to design these patterns in a graphical way, end users must type some EPL code as text. Therefore, these editors require that non-technical users know the corresponding EPL syntax.

On the other side, there are other editors for defining these *situations of interests* as rules. For instance, IBM provides the Operational Decision Management editor (IBM, 2013) which enables end users to write and edit event-condition-action rules in natural language. JBoss proposes Drools Fusion (JBoss, 2013), a module for describing textually inference rules which provides temporal logic analysis. In addition, ALERT (Active support and reaL-time coordination based on Event pRocessing in open source software developmenT), a project supported by the EU 7th Framework Programme, has developed the PANTEON editor (ALERT, 2013). This editor allows to transform graphical event patterns into the EPL provided by the ETALIS engine. Nevertheless, PANTEON only provides Filter, And, Or, Not and Seq operators, while our editor offers much more operators. On the other hand, SocEDA's editor only includes Join, Aggregate and Query into its tool palette.

Finally, there are other editors which allow to define event patterns by means of imperative EPLs. Comparing imperative languages to others, these languages express *how* to do the sequences of actions to be taken, and not only *what* to do. Anyway, they also have limitations regarding their graphical interfaces. Some of these CEP systems are Progress Apama (Progress Software, 2013) and Aurora/Borealis (Abadi et al., 2003).

11. Conclusion and future work

In this paper, we have proposed a model-driven approach for describing complex event patterns in a user-friendly way. In concrete, domain experts, which have a wide expertise knowledge but not EPL knowledge, will be able to make use of our developed editor to easily design the critical or relevant situations (event patterns) that have to be detected in real time. Afterwards, these patterns will be automatically validated and transformed into the EPL code required by the software CEP engine. Thanks to this engine, domain experts will be warned about the situations in which they are interested in, obtaining more valuable information from huge amounts of heterogeneous data shared, processed and stored by many IT systems every day.

In order to show its usefulness, our approach has been applied to a case study for detecting influenza outbreaks around the world. This case study highlights that experts on health care, though not necessarily on CEP, can easily design event patterns for detecting such situations with success using our editor. We have also evaluated and demonstrated that our editor permits the graphical definition of existing event patterns proposed by other authors for other domains. Furthermore, we have analysed the differences among the most relevant EPL languages and proved that our metamodel is valid for any of them.

As a result, we can confirm that our approach, and thereby our proposed metamodel and implemented user-friendly editor, are independent of both the domain where CEP technology is needed to be applied to, and the concrete EPL required by a particular CEP engine.

In our future work, we plan to extend this editor allowing to transform the event pattern models into other EPLs and make an empirical study of the editor use by domain experts, such as health workers or business brokers.

Acknowledgements

This work was funded by the Spanish Ministry of Science and Innovation under the National Program for Research, Development and Innovation, project MoD-SOA (TIN2011-27242). We would like to specially thank Víctor Ayllón and Juan Manuel Reina, Novayre managers, for their fruitful comments and discussions on the topic dealt with in this paper.

References

Abadi, D. J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S., Aug. 2003. Aurora: a new model and architecture for data stream management. The VLDB Journal 12 (2), 120–139.

ALERT, 2013. PANTEON - Interaction Pattern Editor. `http://www.alert-project.eu/content/interaction-pattern-editor`, accessed: 15/06/2013.

Chaudhuri, S., Dayal, U., Narasayya, V., Aug. 2011. An overview of business intelligence technology. Communications of the ACM 54 (8), 88–98.

Cugola, G., Margara, A., Jun. 2012. Processing flows of information: From data stream to complex event processing. ACM Computing Surveys 44 (3), 15:1–15:62.

Dunkel, J., Fernández, A., Ortiz, R., Ossowski, S., Jun. 2011. Event-driven architecture for decision support in traffic management systems. Expert Systems with Applications 38 (6), 6530–6539.

Edge, M. E., Falcone Sampaio, P. R., Sep. 2012. The design of FFML: a rule-based policy modelling language for proactive fraud management in financial data streams. Expert Systems with Applications 39 (11), 9966–9985.

Epsilon, 2013. EuGENia. `http://www.eclipse.org/epsilon/doc/eugenia/`, accessed: 15/06/2013.

EsperTech Inc., 2013. Esper - Complex Event Processing. `http://esper.codehaus.org`, accessed: 15/06/2013.

ETALIS, 2013. ETALIS: Event-driven Transaction Logic Inference System. `https://code.google.com/p/etalis/`, accessed: 17/06/2013.

Etzion, O., Niblett, P., 2010. Event Processing in Action. Manning Publications Co., USA.

Event Processing Technical and Society, Jul. 2011. Event processing glossary - version 2.0. `http://www.complexevents.com/wp-content/uploads/2011/08/EPTS_Event_Processing_Glossary_v2.pdf`, accessed: 15/06/2013.

Hansen, D., Mar. 2013. Big data gets real-time: Oracle fast data. An oracle white paper, Oracle.

Hussmann, H., Meixner, G., Zuehlke, D. (Eds.), Jan. 2011. Model-Driven Development of Advanced User Interfaces. No. 340 in Studies in Computational Intelligence. Springer Berlin Heidelberg.

IBM, 2013. IBM Operational Decision Management. http://www-03.ibm.com/software/products/us/en/subcategory/SW55A, accessed: 15/06/2013.

JBoss, 2013. JBoss Drools Fusion. http://www.jboss.org/drools/drools-fusion.html, accessed: 15/06/2013.

Kolovos, D., Rose, L., García-Domínguez, A., Paige, R., May 2013. The Epsilon Book. http://eclipse.org/epsilon/doc/book/.

Longo, D., Fauci, A., Kasper, D., Hauser, S., Jameson, J., Loscalzo, J., 2012. Harrison's Principles of Internal Medicine: Volumes 1 and 2, 18th Edition. McGraw-Hill, USA.

Luckham, D., 2002. The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley, MA, USA.

Luckham, D., 2012. Event Processing for Business: Organizing the Real-Time Enterprise. Wiley, USA.

Oracle, 2013. Oracle Event Processing. http://www.oracle.com/technetwork/middleware/complex-event-processing, accessed: 15/06/2013.

Progress Software, 2013. Progress Apama Event Processing Platform. http://www.progress.com/es-es/apama/complex-event-processing.html, accessed: 15/06/2013.

Romero, D., Hermosillo, G., Taherkordi, A., Nzekwa, R., Rouvoy, R., Eliassen, F., 2011. The DigiHome service-oriented platform. Software: Practice and Experience.

Russom, P., 2011. Big data analytics. Tech. Rep. 4th Quarter, The Data Warehousing Institute.

SocEDA, 2013. SocEDA - CEP Editor. https://research.linagora.com/display/soceda/CEP+Editor, accessed: 15/06/2013.

Stahl, T., Voelter, M., Czarnecki, K., 2006. Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons.

StreamBase, 2013. StreamBase Studio. http://www.streambase.com/products/streambasecep/streambase-studio, accessed: 15/06/2013.

Sybase, 2013. SAP Sybase Event Stream Processor. http://www.sybase.com/products/financialservicessolutions/complex-event-processing, accessed: 15/06/2013.

Tsuchiya, S., Sakamoto, Y., Tsuchimoto, Y., Lee, V., 2012. Big data processing in cloud environments. Fujitsu Scientific and Technical Journal 48 (2), 159–168.

Uhm, Y., Lee, M., Hwang, Z., Kim, Y., Park, S., Sep. 2011. A multi-resolution agent for service-oriented situations in ubiquitous domains. Expert Systems with Applications 38 (10), 13291–13300.

Vikhorev, K., Greenough, R., Brown, N., Mar. 2013. An advanced energy management framework to promote energy awareness. Journal of Cleaner Production 43, 103–112.

Vincent, P., 2010. The return of the expert system? http://www.thetibcoblog.com/2010/03/12/the-return-of-the-expert-system, accessed: 24/06/2013.

WHO, 2013. World health organization. http://www.who.int/en/index.html, accessed: 15/06/2013.

Yao, W., Chu, C.-H., Li, Z., May 2011. Leveraging complex event processing for smart hospitals using RFID. Journal of Network and Computer Applications 34 (3), 799–810.

Yuan, S.-T., Lu, M.-R., Mar. 2009. An value-centric event driven model and architecture: A case study of adaptive complement of SOA for distributed care service delivery. Expert Systems with Applications 36 (2, Part 2), 3671–3694.