

ModeL4CEP: Graphical domain-specific modeling languages for CEP domains and event patterns

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

Department of Computer Science and Engineering, University of Cádiz, Avda. de la Universidad de Cádiz n° 10, 11519, Puerto Real, Cádiz, Spain

Abstract

Complex event processing (CEP) is a cutting-edge technology that allows the analysis and correlation of large volumes of data with the aim of detecting complex and meaningful events through the use of event patterns, as well as permitting the inference of valuable knowledge for end users. Despite the great advantages that CEP can bring to expert or intelligent business systems, it poses a substantial challenge to their users, who are business experts but do not have the necessary knowledge and experience using this technology. The main problem these users have to face is precisely hand-writing the code for event pattern definition, which requires them to implement the conditions to be met to detect relevant situations for the domain in question by using a particular event processing language (EPL). In order to respond to this need, in this paper we propose both a graphical domain-specific modeling language (DSML) for facilitating CEP domain definitions by domain experts, and a graphical DSML for event pattern definition by non-technological users. The proposed languages provide high expressiveness and flexibility and are independent of event patterns and actions' implementation code. This way, domain experts can define the relevant event types and patterns within their business domain, without having to be experts on EPL programming, nor on other complicated computer science technological issues, beyond an understandable and intuitive graphical definition. Furthermore, with these DSMLs, users will also be able to define the actions to be automatically taken once a pattern is detected in the system. Further benefits of these DSMLs are evaluated and discussed in depth in this paper.

Keywords: Complex Event Processing, Model-Driven Development, Domain-Specific Modeling Language, Event Processing Language.

* Corresponding author. Tel.: +34 956 483482

Email addresses: juan.boubeta@uca.es (Juan Boubeta-Puig), guadalupe.ortiz@uca.es (Guadalupe Ortiz), inmaculada.medina@uca.es (Inmaculada Medina-Bulo)

Publisher version:

Boubeta-Puig, J., Ortiz, G., & Medina-Bulo, I. (2015). ModeL4CEP: Graphical domain-specific modeling languages for CEP domains and event patterns. *Expert Systems with Applications*, 42(21), 8095–8110. <https://doi.org/10.1016/j.eswa.2015.06.045>

1. Introduction

Complex Event Processing (CEP) (Luckham, 2002) is an emerging technology that allows us to process, analyze and correlate a huge amount of heterogeneous data in form of events with the aim of detecting relevant or critical situations for a particular domain in real time. For that purpose, the conditions describing situations to be detected must be specified by using special templates known as *event patterns*. These patterns are added into a CEP engine, the software responsible for analyzing and correlating received events, 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), and allow us to implement the decision model for an expert or intelligent business system, which is in charge of taking appropriate actions on demand.

CEP has already been applied to several domains such as health care (Boubeta-Puig, Ortiz, & Medina-Bulo, 2011, 2014a; Yuan & Lu, 2009), home automation (Boubeta-Puig, Ortiz, & Medina-Bulo, 2014b; Romero et al., 2011), network analysis and surveillance (Gad, Boubeta-Puig, Kappes, & Medina-Bulo, 2012), maritime traffic management (Boubeta-Puig, Medina-Bulo, Ortiz, & Fuentes-Landi, 2012), location-based services (Uhm, Lee, Hwang, Kim, & Park, 2011), operational intelligence in business (Chaudhuri, Dayal, & Narasayya, 2011), and transportation and traffic management (Dunkel, Fernández, Ortiz, & Ossowski, 2011).

According to Vincent (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 this scope, even though a diversity of domains can currently benefit from CEP technology, as mentioned before, the main handicap for subject matter experts is the need to define the event types for a concrete domain in the EPL syntax provided by the CEP engine to be used in the system in question. This has two major drawbacks: on the one hand, a domain of interest should be implemented as many times as EPLs are to be used. This implies an increase of time required to define the CEP domain as well as more expensive maintenance, since a modification in the domain –adding a new event type, for example– will require a modification in each implementation related to the same domain. On the other, since any domain definition will have to be implemented manually, developing this task will be challenging for non-computer users.

In this regard, there are approaches (Bruns, Dunkel, Lier, & Masbruch, 2014; Bruns, Dunkel, Masbruch, & Stipkovic, 2015a) which permit specifying what the domain of each event is. Nevertheless, none of them offers the possibility to describe a CEP domain composed of a set of event types together with their domain description. The lack of these elements in the domain will prevent the chance of sharing the domains among different modeling tools, which might belong to different users.

To solve these problems, we propose a graphical Domain-Specific Modeling Language (DSML) that provides domain experts with an intuitive and user-friendly way for defining the CEP domains of interest for which they need to detect critical situations in real time. This will facilitate the unification of CEP domain descriptions, represented as models, which might be shared by different experts.

Besides, in order to detect real-time situations using CEP technology, it is necessary to implement event patterns using the EPL provided by the CEP engine in question. To do this, high expertise in that language is needed. This causes users who have a vast knowledge of the domain for which they need to define event patterns but are inexperienced in CEP to be unable to define those event patterns themselves. In this regard, a survey by ebizQ (BEA, 2007) concluded that 84% of respondents consider that event pattern definition should be performed by domain experts, who really have all the necessary knowledge for this, compared to 16% who think it is more convenient to leave this task to programmers.

Although there are several approaches to event pattern definition (Stühmer et al., 2009; Yao, Chu, & Li, 2011), among others, they are often oversimplified, providing only a reduced set of operators and data windows. Thus, they have limitations regarding the expressiveness of complex patterns to be modeled and hinder scalability of the defined event patterns.

Given these problems and the amount of existent EPLs for implementing event patterns, we also propose a graphical DSML in this paper, with the aim of creating a “common language” so that any user can easily define a pattern, regardless of the language required to implement and deploy the pattern in the CEP engine. Thanks to Model-Driven Development (MDD) techniques, this pattern defined as a model will be transformed into different EPLs with the following consequent advantages: EPL technical aspects are hidden from end users, and productivity and software quality

improve since models are easier to maintain; furthermore, the automatic generated code will be error-free.

In our previous work (Boubeta-Puig et al., 2014a), we defined a model-driven approach for facilitating user-friendly design of complex event patterns. To support this, a first version of a DSML for event pattern definition was proposed. However, this DSML, composed of an abstract syntax –an EPL metamodel along with its restrictions– and a concrete syntax establishing the relationship between the metamodel concepts and their graphical representation, presented several limitations compared to our novel DSMLs proposed in this paper. First of all, a DSML for CEP domain definition was not considered in our previous work, lacking the above mentioned advantages. Secondly, the graphical notation of the concrete syntax was not as user-friendly. Thirdly, the EPL metamodel was incomplete and less understandable by end users, as will be discussed throughout this paper.

Therefore, this work’s main contributions are two unprecedented and highly expressive DSMLs for CEP domain and event pattern definition, including their graphical notation, bringing CEP closer to any user, thus facilitating expert system management to business experts, who master the business in question but certainly do not need to have in deep knowledge of expert or intelligent systems. Thanks to the contributed DSMLs, the business expert will be able to easily define the event types and patterns relevant to a particular intelligent system and the latter would provide us with the suitable action to be taken, without the need of learning any programming language, in particular a CEP-based one.

The rest of this paper is organized as follows: Section 2 includes background on MDD and CEP, while Section 3 describes related work. Sections 4 and 5 describe the proposed DSMLs for CEP domain and event pattern definition, respectively. Then, these languages are evaluated and discussed in Section 6. Finally, conclusions and future work are highlighted in Section 7.

2. Background

In this section, the subject matters relevant to the scope of this paper, MDD and CEP, are introduced.

2.1. Model-driven development

Currently, the software engineering community is more oriented towards the development of applications based on models rather than technologies. MDD focuses on essential aspects of the system, postponing the choice of the most appropriate technology for implementation (Ortiz, 2007). Model definition and transformations between models and models to code are key parts for developing automatic systems. The high reusability and reliability of the code generated by this software paradigm, as well as the increased productivity and less costly maintenance, have led to its application in various fields, trying to satisfy different needs in both the business world and the academic community (García-Molina, 2013; Stahl, Voelter, & Czarnecki, 2006).

In this scope, a model is a simplified representation of a given reality with the aim of understanding it better. To achieve this abstraction, irrelevant details of this reality must be removed from the model regardless of whether it is represented using textual or graphical notation. A graphical representation of a model is called a diagram.

Models are created by using DSMLs, whose definition consists of three distinct parts: 1) the abstract syntax that consists of both a metamodel –a model describing language concepts and relationships between them– and validation rules to check whether a model is well formed, 2) the concrete syntax or DSML notation –the set of useful graphical symbols for drawing diagrams–, and 3) transformations between models and model to code for software automation. Note that a model, i.e. an instance of a metamodel, may have different graphical notations. According to Fowler (Fowler & Parsons, 2010), some advantages of using this language type are: development productivity advancement, improved communication with domain experts, easier adaptation to changes, and users help in the task of specifying what the system should do, but not on how it should be conducted.

A metamodeling language is required for creating a model. This language also has both an abstract syntax and a concrete syntax. The abstract syntax is defined by a meta-metamodel, i.e., using the metamodel language itself, whereas the concrete syntax can be defined either by graphical notations, such as Unified Modeling Language (UML) class diagrams (OMG, 2014), or by textual notations, such as Emfatic (Eclipse Foundation, 2012), a textual notation to define Ecore metamodel. Ecore is currently one

of the most used metamodeling languages and it is part of the Eclipse Modeling Framework (EMF) metamodel architecture (Steinberg, Budinsky, Paternostro, & Merks, 2008).

Some authors (Atkinson et al., 2001; Kleppe, Warmer, & Bast, 2003) have proposed a four-level architecture (see Figure 1) to explain the relationship between models and metamodels, as detailed below:

- Data level (M0): it represents real-world data that conforms to a given model.
- Model level (M1): it characterizes models describing M0-level data. Every model conforms to a metamodel.
- Metamodel level (M2): it characterizes metamodels describing M1-level models. Every metamodel conforms to a meta-metamodel.
- Meta-metamodel level (M3): it characterizes meta-metamodels describing M2-level metamodels. A meta-metamodel conforms to itself.

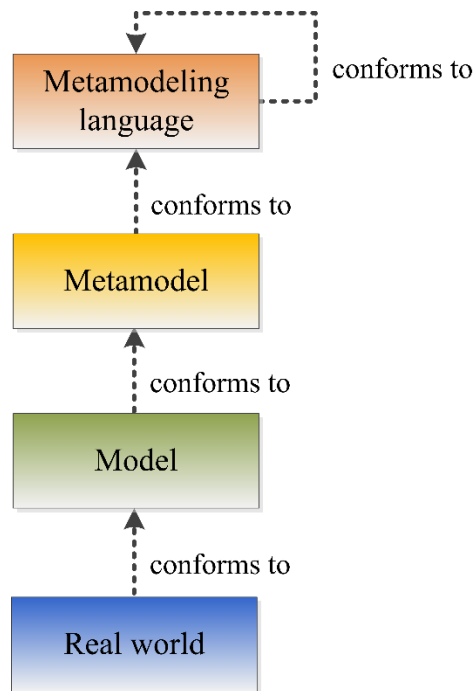


Figure 1. Four-layer modeling architecture.

2.2. Complex event processing

A CEP overview, together with its advantages as well as the existing language types for using this technology are detailed in the following subsections.

2.2.1. *Fundamentals*

As previously mentioned, CEP is an emerging technology that allows us to analyze and correlate huge amounts of data in form of events with the aim of detecting relevant or critical situations (complex events) in real time.

An event can be defined as anything that happens or could happen (Luckham, 2012), but also anything that could happen but does not happen. A situation is an event occurrence or an event sequence that requires an immediate reaction (Etzion & Niblett, 2011). Events can be classified into three main 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 Society, 2010). Events can be derived from other events by applying or matching event patterns, templates where the conditions describing situations to be detected are specified. A CEP engine is the software used to match these patterns over continuous and heterogeneous event streams (timely ordered sequence of events of multiple types), and to raise alerts about complex events created when detecting such event patterns.

As depicted in Figure 2, CEP is performed in three stages:

- Event capture: it consists of the reception of events to be analyzed by CEP technology.
- Analysis: from the event patterns previously defined in the CEP engine, it will process and correlate the information in the form of events in order to detect critical or relevant situations in real time.
- Response: after detecting a concrete situation, it will be notified to the system, software or device in question.

The main advantage of using this technology to process complex events is that these can be identified and reported in real time, thus reducing latency in decision making, unlike the methods used in traditional software for event analysis. According to (Chandy & Schulte, 2010), other important advantages are: decision quality improvement, faster and (semi-)automatic reply, information overload prevention and human workload reduction.

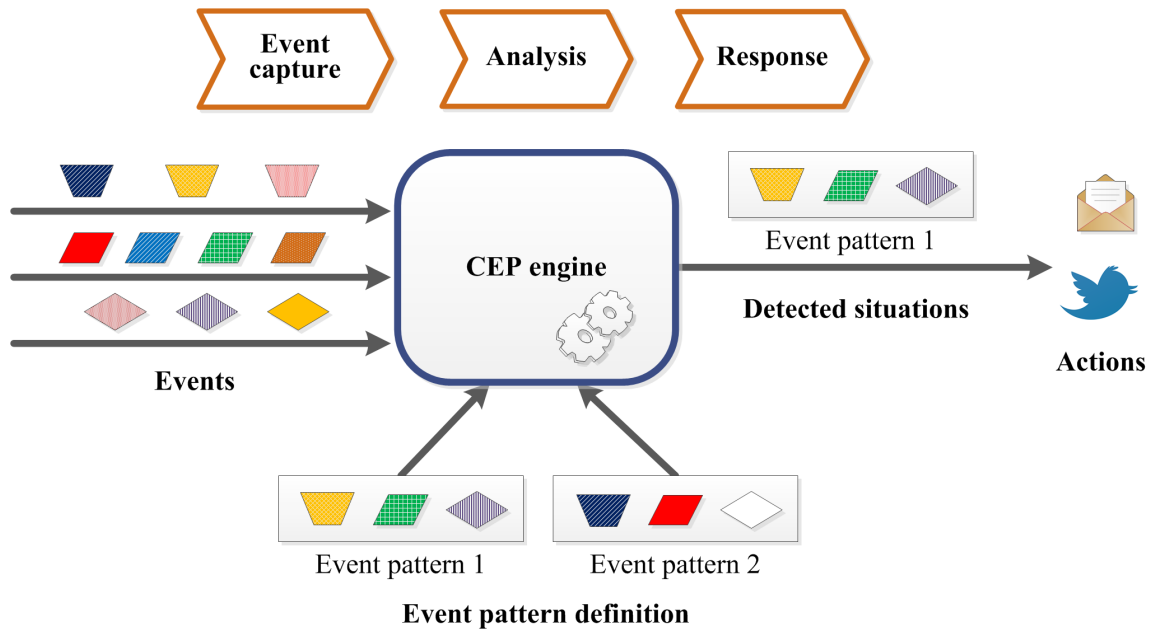


Figure 2. Complex event processing stages.

2.2.2. Event processing languages

As already mentioned, in order to detect situations of interest on specific domains it is necessary to define so-called event patterns. These are defined by using specific languages developed for this purpose, known as EPLs. These languages can be classified by the following language styles (Etzion & Niblett, 2011): stream-oriented, rule-oriented and imperative.

Stream-oriented EPLs are SQL-like languages and algebra relational languages which include new concepts, such as temporal relationships and data windows. Some of these EPLs are: Esper EPL (EsperTech, 2015), Oracle EPL (Oracle, 2015), StreamSQL (TIBCO, 2015) and CCL (Sybase, 2015).

Rule-oriented EPLs are classified into three subtypes: production rules, active rules and logic-programming rules.

Production rules are *if-condition-action* type, i.e., when the condition is satisfied, then the action is executed. Drools Fusion (JBoss Community, 2014) is one of most well-known languages in this category.

Active rules, also known as *Event-Condition-Action* (ECA), are based on active databases characterized by the following behavior: when an event happens, its

conditions are evaluated and, if these are satisfied, then the corresponding action is carried out. IBM Operational Decision Management (IBM, 2014) provides one of these languages.

Logic-programming rules are based on logic assertions and deductive databases, allowing deductions through inferences from rules and a set of facts. ETALIS (Anicic & Fodor, 2014) is a CEP solution implemented in Prolog that provides two rule-based languages: ELE (ETALIS Language for Events) and EP-SPARQL (Event Processing SPARQL).

Finally, imperative EPLs define rules in an imperative way where operators define transformations over their inputs. Progress Apama (Software AG, 2014) is an event processing platform that provides this language style.

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

3. Related work

In recent years, various approaches have been proposed to define application domains as well as event patterns in fields where CEP needs to be applied. Some of these use ontological languages for such definitions. Other approaches employ MDD techniques in order to define domains and event patterns as models, which will then be transformed into the specific code required by the CEP system to be used.

In the following subsections, these ontological and model-driven approaches are detailed.

3.1 Ontological approaches

Several works using ontologies for the representation of events and event patterns have been found.

Sen et al. (Sen & Stojanovic, 2010; Sen, Stojanovic, & Stojanovic, 2010) have proposed a semantic model for the representation of events and event patterns based on Resource Description Framework Schema (RDFS) (W3C, 2014b), a language for knowledge representation that provides the necessary elements for ontology description. In particular, the defined ontology contains a set of concepts (*Event*, *EventOperator*, *EventSource* and *EventType*) and a set of properties (*hasStartTime*, *hasEndTime*,

hasEventName, *hasEventId*, *hasEventSource*, *hasEventType* and *composedOf*). The *EventType* and *EventSource* concepts allow us to classify events with similar characteristics depending on the type and source from which they come. Each event has a unique identifier (*hasEventId*) and a name (*hasEventName*). Whereas a simple event is indivisible and occurs at a particular time (*hasStartTime*), a complex event can occur over a period of time between start date (*hasStartTime*) and end date (*hasEndTime*). Each complex event, being composed of other events, will additionally have a property called *composedOf*. It is noteworthy that event operators considered in this ontology are the same as those proposed by Chakravarthy and Mishra (Chakravarthy & Mishra, 1994), i.e., aggregation operator (*COUNT*), data windows (*WITHIN*), logical operators (*AND* and *OR*) and temporal operator (*SEQ*). In this semantic model, an event pattern is described as a set of events and event operators.

This model proposed by Sen et al. for event and event pattern representation using ontologies has been used with some modifications in the ALERT European project (Active support and real-time coordination based on Event Processing in FLOSS development) (ALERT, 2013), which belongs to FP7 (the Seventh Framework Programme for Research and Development). Specifically, a new concept called “interaction pattern” has been added to such model, extending the event representation by a set of properties that facilitate the event pattern definition: *identifier*, *name* and *pattern creation date*. As in the previous model, aggregation operators, data windows, logical operators and temporal operators are included.

Stühmer et al. (Stühmer et al., 2009) also present an RDFS ontology for events, where an event can be represented as a simple or as a complex event. All events have a *type*, *start timestamp*, *end timestamp* and, optionally, the *message payload*. Unlike other proposals, a complex event is specialized by event operator types needed to define concrete situations to be detected, such as *AndEvent*, *OrEvent* or *NotEvent*.

Paschke et al. (Paschke, Boley, Zhao, Teymourian, & Athan, 2012; Paschke, 2014) have created the Reaction RuleML model, which defines the following ontological structure of concepts: event, situation, space, time, action and agent. These concepts can be related to each other and also specialized by existing domain ontologies –vocabularies related to a specific domain– and by ontologies for generic tasks and activities, such as processing situations of interest. Each situation of interest will be composed of its properties (*hasProperties*) and a description (*hasContent*) and will also

belong to one of the following categories: heterogeneous –situations which are influenced by dynamic changes, time or frequency– or homogeneous –stable, iterative or usual situations–.

Before defining the Reaction RuleML model, Paschke published a semantic design pattern language for CEP (Paschke, 2009) on his own. This has been implemented as an event pattern description language based on XML, an English natural language description and an ontological language implemented as an OWL (Web Ontology Language) vocabulary language (W3C, 2014a).

Yao et al. (Yao et al., 2011) have also proposed a CEP ontology that consists of events classified as simple and complex events. Each event has a type and a set of attributes. Simple events are also classified into RFID (Radio Frequency IDentification) events and non RFID events. Moreover, complex events are composed of event operators: logical and temporal operators. This ontology has been already applied to health domains in which all information about a hospital has been monitored by means of RFID devices.

All the analyzed ontological approaches have limitations compared to the model-driven approach proposed in this paper. On the one hand, these ontological approaches provide an incomplete set of operator types and data windows. This fact implies a limitation in the expressiveness of complex patterns to be modeled. Furthermore, the way in which some of these ontological models have been structured hinders scalability and understanding of the defined event patterns. For example, Stühmer’s model represents operators as classes specialized of *ComplexEvent* class, instead of creating an operator class that generalizes the different operator types, and the arity operator is not determined.

3.2 Model-driven approaches

The existing model-driven approaches have been classified into two categories – textual and graphical– according to the notation used to represent event patterns.

Regarding approaches that provide a textual representation for event patterns, there are several which deserve a special mention and are described below.

Zang et al. (Zang, Fan, & Liu, 2008) have defined a metamodel in which events are interrelated with event operators, processes and contexts. These events can come

from various sources, such as services, databases, RFID devices and process activities. Just as in previous proposals, event types are classified into simple and complex events, which may be connected through causal relationships. This metamodel defines the following operator types: logical, temporal, causal and RFID. These enable the combination of events in order to create situations (complex events). Moreover, they may be part of a context through which an event hierarchy could be created.

Cugola et al. (Cugola & Margara, 2010) have described TESLA, a complex event specification language composed of event and event pattern textual models. It provides temporal and content constraints, parameterization, negations, sequences, iterations, aggregates and timers, as well as the possibility of combining event patterns to create hierarchies of events. The author propose that TESLA event patterns can be translated into automata. Subsequently, they have presented a new model called CEP2U for dealing with uncertainty in CEP (Cugola, Margara, Matteucci, & Tamburrelli, 2015). Two types of uncertainty are tackled in this work: uncertainty in the data coming from sources (events) and uncertainty in event patterns. The uncertainty of event patterns are modeled through Bayesian networks.

There are other model-driven approaches for CEP providing textual languages that have been implemented in order to define event patterns restricted to a particular domain. Mulo et al. (Mulo, Zdun, & Dustdar, 2013) have proposed an approach for compliance monitoring in process-driven service-oriented architectures. Concretely, they define a textual DSML for the specification of compliance directives and use model-to-text transformations to generate compliance monitoring code.

Bruns et al. (Bruns et al., 2014) have defined a textual DSML called DS-EPL (Domain-Specific Event Processing Language) with the aim of defining event patterns on a specific domain, namely, a language for modeling event patterns in machine-to-machine (M2M) domain. Furthermore, the authors have evaluated this DSML, applying it to a solar power plant case study. In this approach, a new DSML would be required for each domain where CEP needs to be applied, causing increased workload and additional effort for each new language to be implemented. To solve this problem, we propose an event pattern DSML independent of the domain where CEP needs to be applied, providing the feature of being customized to any CEP domain based on the event types modeled for the new domain to be incorporated. Besides, DS-EPL provides a predefined set of event types that can be extended in order to define new ones;

however, the possibility of creating other event types independent of these predefined ones is not considered. Furthermore, actions to be executed when detecting situations of interest are not supported.

Afterwards, Bruns et al. (Bruns, Dunkel, Masbruch, & Stipkovic, 2015b) have defined models for the M2M domain, M2M machine states and M2M events based on DS-EPL. Therefore, this proposal has the same aforementioned limitations found in the previous work, such as the difficulty for non-technical users to define event patterns by using a textual notation, except for event patterns actions that are supported in this new version.

Terroso-Saenz et al. (Terroso-Saenz et al., 2015) have also defined an event model whose root element is the *RootEvent*. This is specialized by *LocationEvent*, *MovementEvent*, *RelationshipEvent* and *MovementChangeEvent*. In particular, this proposal aims to analyze entities' trajectory data in real time. According to the authors, an entity's trajectory is defined as the stream of location events related to it. A new textual notation is proposed to define event patterns; this model is dependent on the trajectory analysis domain.

On the other hand, Terroso-Saenz et al. (Terroso-Sáenz, Valdés-Vela, Campuzano, Botia, & Skarmeta-Gómez, 2015) have created a CEP approach to perceive the vehicular context. Specifically, they have proposed an intra-vehicular context information model and a set of event patterns to detect the vehicular occupancy. One of the main advantages of this approach is the use of Markov models to predict the next destination of a person or vehicle, so the probability of taking appropriate actions is considered in this work.

Although most of aforementioned approaches allow us to specify what the domain of every event is, none of them offers the possibility of describing a CEP domain composed of a set of event types together with their domain description. The lack of these elements in the domain will prevent sharing the domains among different modeling tools, which might belong to different users. Unlike the last mentioned approaches, our approach is designed to be independent from application domains, event languages and event processing systems.

Regarding approaches that provide a graphical representation for event patterns, there are two metamodels, proposed by Obweger et al. and Etzion and von Halle, which deserve a special mention and are described below.

Obweger et al. (Obweger, Schiefer, Kepplinger, & Suntinger, 2010; Obweger, Schiefer, Suntinger, & Kepplinger, 2011) have proposed an event model, a correlation model and a rule model, among others. The event model allows us to define the event types to be handled in the system. There are three event types: simple values –number, string and other event types–, data collections and data dictionaries. The correlation model provides the ability to establish relationships between events. By means of the rule model, the situations to be detected (complex events) are described. An event pattern definition using Obweger’s metamodel consists of a set of components, a set of precondition relationships, a set of inputs and a set of outputs. These components allow us to establish the conditions to be met for detecting situations of interest, to specify time intervals, and to indicate whether a pattern depends on another one previously defined. In addition, these components can be linked by connecting the output ports of a component –actions to be taken after detecting a defined situation– to the input ports of another component –preconditions to be met in order to analyze the component content–.

Etzion and von Halle (Etzion & von Halle, 2013) have presented a model-driven approach for defining event patterns known as The Event Model. This approach is based on the concepts described by the main author in (Etzion & Niblett, 2011). According to the authors, the features of this approach are as follows: a structure to carry out a rigorous modeling of the reality is provided, patterns are represented using tables and can be automatically transformed into EPL code, and models are independent of implementation.

The analyzed graphical approaches have used notations for event pattern definition in table and/or textual formats, whereas our approach utilizes graphical nodes and links. The latter are more intuitive for any user, regardless of their skills concerning CEP technology.

There are other model-driven approaches for CEP which provide graphical DSMLs. Nevertheless, these also fall outside the scope of this work because they were created in order to define event patterns restricted to a particular domain. For instance,

Decker et al. (Decker, Grosskopf, & Barros, 2007) have proposed a graphical language, called BEMN (Business Event Modeling Notation), with the aim of modeling event patterns in the context of business processes.

4. Domain-specific modeling language for CEP domain definition

In this section, a DSML for defining CEP domains is proposed, specifying both the abstract and concrete syntax for this language.

4.1. Abstract syntax

The abstract syntax of a DSML is composed of a metamodel, where the language concepts and relationships between them are defined, as well as the restrictions for model elements and their relationships in order to ensure the compliance of domain rules. The next subsection describes the CEP domain metamodel and the rules for checking that domain models are well-formed.

4.1.1 CEP domain metamodel

The proposed metamodel for defining CEP domains is described in detail in this section. Figure 3 shows the metaclasses of this metamodel and their relationships, which are described as follows:

- *CEPDomain*: it is the main metaclass, so the root of every model will be a unique *CEPDomain* instance. This represents a concrete CEP domain composed of one or more event types (*Event*). For every domain, it is necessary to specify its name (*domainName*), a textual description (*domainDescription*) and creation date (*domainCreationDate*).
- *Event*: it describes an event for a concrete CEP domain. Every event has a type (*typeName*) and, additionally, may have an image that represents it graphically, in this case its path (*imagePath*) must be indicated. Furthermore, every event will be composed of one or more event properties (*EventProperty*).
- *EventProperty*: it represents the property or attribute of an event. Every property must have a name (*name*) and one of the following types (*type*): *Unknown*, *Boolean*, *Integer*, *Long*, *Double*, *Float* or *String*. A property may have an image that represents it graphically, in which case its path

(*imagePath*) must be indicated. Moreover, a property can contain one or more properties, i.e., the definition of nested properties is supported.

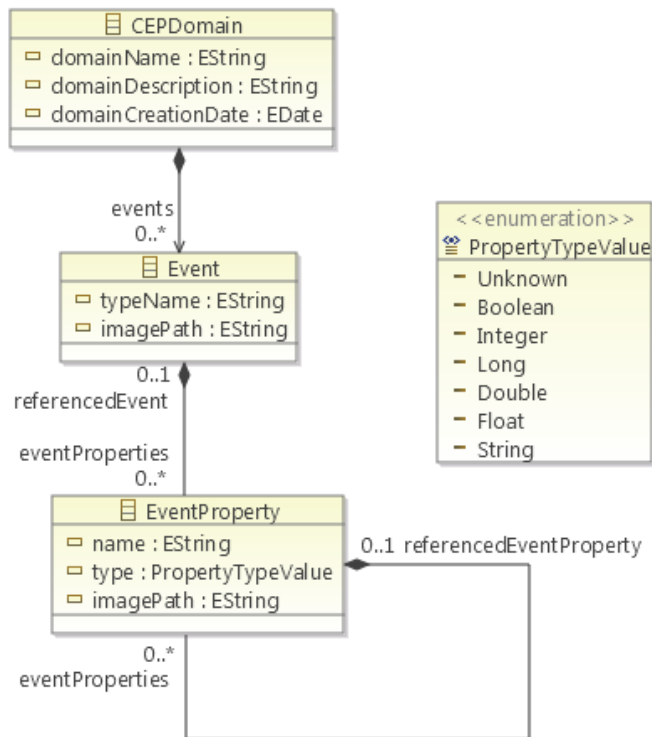


Figure 3. CEP domain metamodel.

4.1.2. Restrictions for CEP domain metamodel

Next, restrictions for CEP domain metamodel are presented. Table 1 shows the validation rules that any model conformed to the metamodel must satisfy, as well as describing restrictions for each metaclass.

Table 1. Restrictions for CEP domain metamodel.

Metaclass	Restriction
<i>CEPDomain</i>	The domain name (<i>domainName</i>) must be specified. It must contain, at least, an event (<i>Event</i>).
<i>Event</i>	The event type name (<i>typeName</i>) must be specified. It must contain, at least, an event property (<i>EventProperty</i>).
<i>EventProperty</i>	The property name (<i>name</i>) must be specified. If it is a nested property, this property cannot contain properties with the same name (<i>name</i>) in the same nested level. If it is a nested property, this property will not be able to have a property type (<i>type</i>), since the type will be determined by its contained properties.

4.2. Concrete syntax

The concrete syntax of a DSML allows us to establish a relationship between metamodel concepts and their textual or graphical representation.

In addition to the definition of both CEP domain metamodel and restrictions, a graphical notation has been created for every element that can be used for designing a CEP domain model. This concrete syntax for CEP domain models is shown in Table 2.

Table 2. Concrete syntax for CEP domain metamodel.

Name	Notation
<i>Event</i>	E
<i>EventProperty</i>	P

5. Domain-specific modeling language for event pattern definition

In this section, a DSML for defining event patterns is proposed, specifying both abstract and concrete syntax for this language.

5.1. Abstract syntax

As previously mentioned, the abstract syntax of a DSML is composed of a metamodel, where the language concepts and relationships between them are defined, as well as the restrictions for model elements and their relationships in order to guarantee the compliance of domain rules. The next subsection describes the event pattern metamodel and the rules for checking that event pattern models are well-formed.

5.1.1 Event pattern metamodel

The metamodel for defining event patterns in a user-friendly and intuitive way is presented in the following lines. Figure 4 shows the main metaclasses of this metamodel and their relationships, described below following top-down and left-right order:

- *CEPEventPattern*: it is the main metaclass of the metamodel, so the root of every model will be an instance of *CEPEventPattern* that represents an event pattern. This pattern can contain links (*Link*) –to establish relationships between the rest of elements–, elements (*EventPatternElement*) –necessary for defining conditions to be detected by the pattern–, a complex event (*ComplexEvent*) –the event type to be

created when detecting the pattern– and actions (*Action*) –to be carried out when detecting the pattern. For every event pattern, it is necessary to specify its name (*patternName*), a textual description (*patternDescription*), the domain to which this pattern belongs (*domainName*), creation date (*patternCreationDate*), and whether the pattern has been already transformed into code and deployed in the CEP engine (*patternDeployed*).

- *Link*: it defines the graphical representation of one or more relationships between operands (*Operand*) and operators (*Operator*). For every link established between an operand and an operator, the order (*order*) of this operand with respect to the rest of operands linked by this operator must be specified. The value of *order* can be set between 1 and N, N being the total number of operands linked by the operator.
- *Operand*: it is a data on which the linked operation is performed. There are condition operands (*ConditionOperand*) –these can be linked by condition operators– and pattern operands (*PatternOperand*) –these can be linked by pattern operators–. Figure 5 illustrates *ConditionOperand* and *PatternOperand* metaclasses, and Table 3 describes the operand types. In order to define more complex event patterns, an operator (*Operator*) can be, at the same time, an operand type, allowing it to be linked by another operator. This is the reason why Figure 4 shows the aggregation operator (*AggregationOperator*) as an operand type. Besides, a complex event (*ComplexEvent*) is also considered as an operand type, since it could be necessary to link it to an action (*Action*), to be carried out on detection.
- *Operator*: it is used to express a specific operation between one or more operands –depending on the arity of the operator. There are condition operators (*ConditionOperator*) –these can be linked by condition operands–, pattern operators (*PatternOperator*) –these can be linked by pattern operands– and aggregation operators (*AggregationOperator*) –aggregation functions that can be applied to some operands–. Figure 6 illustrates *ConditionOperator*, *PatternOperator* and *AggregationOperator*. Notice that the arity of every operator –unary,

binary or n-ary– is not shown for clarity purposes; this information is specified in Table 4, where operator types are described.

- *UnaryOperator*: an operator that must be linked by an operand.
- *BinaryOperator*: an operator that must be linked by two operands.
- *NaryOperator*: an operator that must be linked by two or more operands.
- *EventPatternElement*: it represents the elements to be used for pattern definition. These elements are classified into two types: *EventPatternCondition* –the conditions to be met to detect a concrete event pattern– and *DataWindow* –if the condition search is only applied to a subset of events or under temporal restrictions. *EventPatternCondition* is, at the same time, classified into five categories: *ConditionOperand*, *PatternOperand*, *ConditionOperator*, *PatternOperator* and *AggregationOperator*. Figure 7 shows *DataWindow* metaclass and Table 5 describes the data window types.
- *ComplexEvent*: it describes the complex event type to be created when detecting the pattern. The complex event will have a concrete type (*typeName*) and, additionally, an image that represents it graphically; in this case its location (*imagePath*) must be specified. The type name (*typeName*) will be the same as the event pattern name (*patternName* in *CEPEventPattern*); therefore, the event pattern name will determine which pattern has detected it. Moreover, every complex event will be composed of one or more complex event properties (*ComplexEventProperty*).
- *Action*: it indicates the action to be carried out once a complex event is created when detecting its corresponding event pattern. These actions are classified into two categories: *Email* –to send a complex event by email– and *Twitter* –to send it to a Twitter account. Figure 8 illustrates *Action* metaclass and Table 6 describes action types. Although only two action types have been defined so far, this metamodel can be easily extended with new actions, should it be necessary. In order to do that, a new metaclass which extends to *Action* metaclass must be created per action to be added.

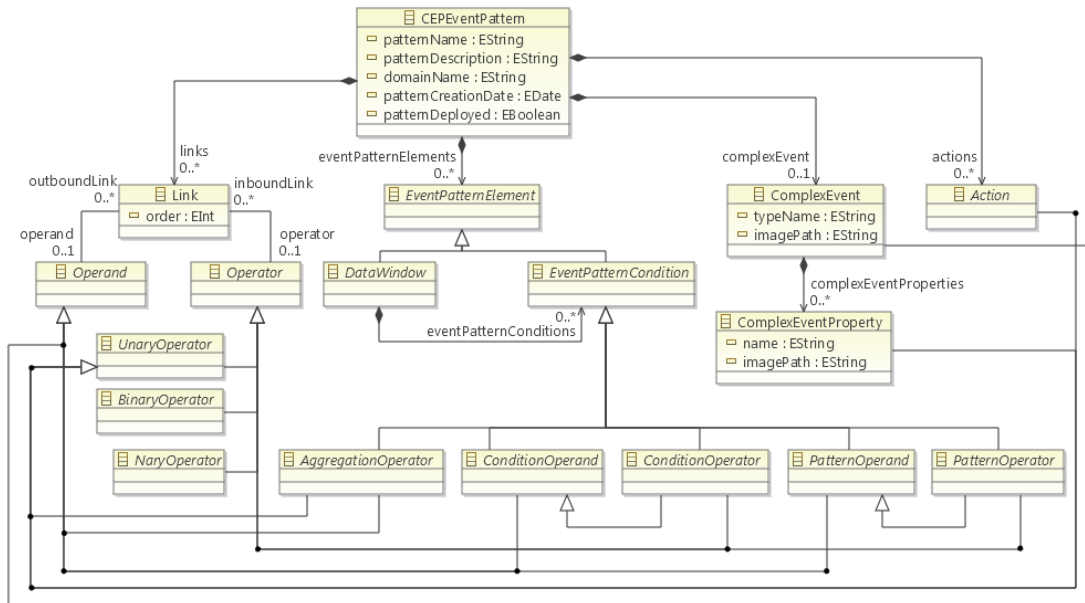


Figure 4. Event pattern metamodel.

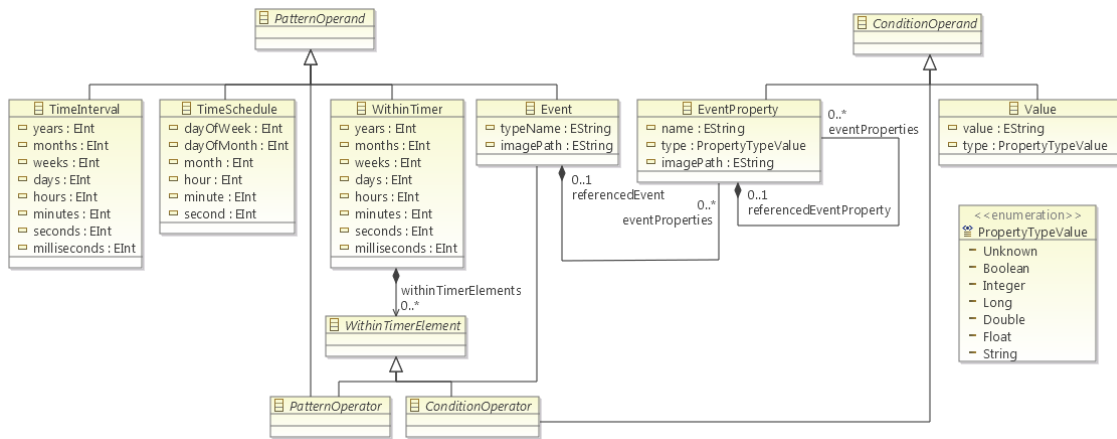


Figure 5. Event pattern metamodel: operands.

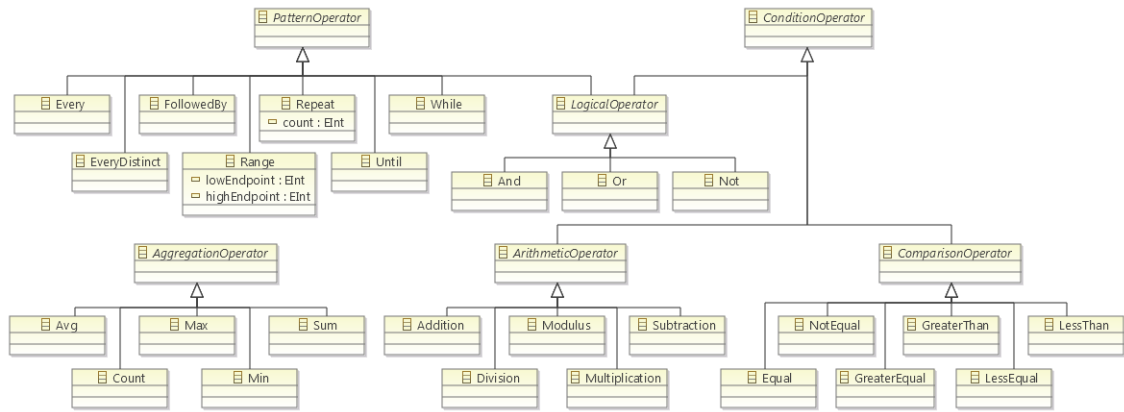


Figure 6. Event pattern metamodel: operators.

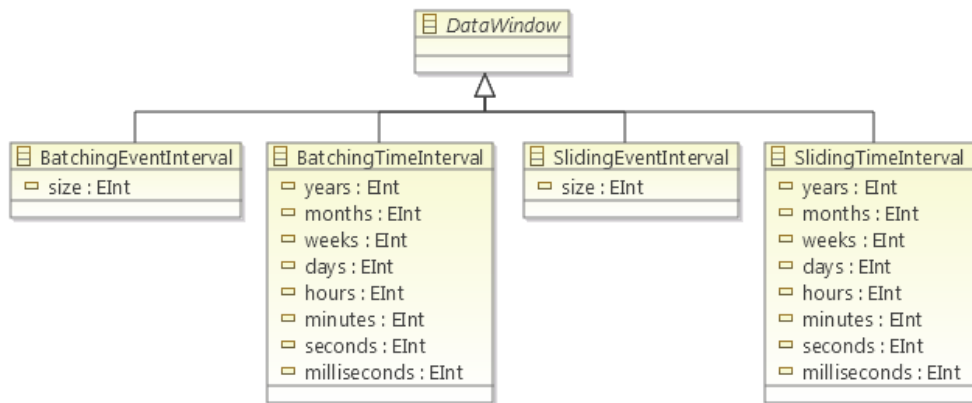


Figure 7. Event pattern metamodel: data windows.

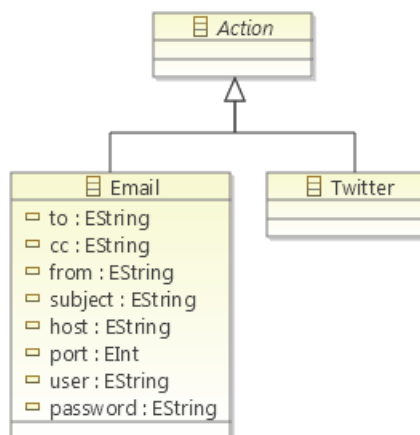


Figure 8. Event pattern metamodel: actions.

Table 3. Operands of the event pattern metamodel.

Type	Operand	Description
Pattern	<i>TimeInterval</i>	It waits for the specified time period (<i>years, months, weeks, days, hours, minutes, seconds</i> and <i>milliseconds</i>) before turning to true.
	<i>TimeSchedule</i>	It turns into true at a defined time (<i>dayOfWeek, dayOfMonth, month, hour, minute, second</i>).
	<i>WithinTimer</i>	It is permanently evaluated to false if the contained pattern expression does not turn to true during the specified time period (<i>years, months, weeks, days, hours, minutes, seconds</i> and <i>milliseconds</i>).
Pattern & WithinTimer Element	<i>Event</i>	It describes an event for a concrete CEP domain. Every event must have a type name (<i>typeName</i>) and may have an image representing it graphically, whose location (<i>imagePath</i>) must be specified. Moreover, every event is composed of one or more event properties (<i>EventProperty</i>).
Condition	<i>EventProperty</i>	A property describes a feature of an event. Every property must have a name (<i>name</i>) and one of the following types: <i>Unknown, Boolean, Integer, Long, Double, Float</i> or <i>String</i> . The property may have an image which represents it graphically, whose location (<i>imagePath</i>) must be specified. Moreover, a property can, at the same time, contain one or more properties, i.e., nested properties are supported.
	<i>Value</i>	It defines a <i>Boolean, Integer, Long, Double, Float</i> or <i>String</i> value.

Table 4. Operators of the event pattern metamodel.

Type	Subtype	Operator	Arity	Description
Pattern		<i>Every</i>	U	It selects every event belonging to the specified type.
		<i>EveryDistinct</i>	U	It is similar to <i>Every</i> , but eliminates duplicated results according to a given <i>distinct-value</i> expression.
		<i>FollowedBy</i>	N	It determines a pattern expression that must be followed by another.
		<i>Range</i>	U	It specifies the minimum (<i>lowEndpoint</i>) and maximum (<i>highEndpoint</i>) number of times a pattern expression must occur.
		<i>Repeat</i>	U	It defines how many times (<i>count</i>) a pattern expression must occur.
		<i>Until</i>	B	It checks a pattern expression until the condition (another pattern expression) is evaluated to true.
		<i>While</i>	B	It checks a pattern expression while the condition (another pattern expression) is evaluated to true.
Pattern & Condition	<i>Logical</i>	<i>And</i>	N	It returns a true value only if all operands are true.
		<i>Or</i>	N	It returns a true value if at least one operand is true.

	<i>Not</i>	U	It returns a true value if the operand is false, and a false value if the operand is true.	
<i>Condition</i>	<i>Comparison</i>	<i>Equal</i>	B	It returns a true value if operand1 = operand2.
		<i>GreaterEqual</i>	B	It returns a true value if operand1 >= operand2.
		<i>GreaterThan</i>	B	It returns a true value if operand1 > operand2.
		<i>LessEqual</i>	B	It returns a true value if operand1 <= operand2.
		<i>LessThan</i>	B	It returns a true value if operand1 < operand2.
		<i>NotEqual</i>	B	It returns a true value if operand1 ≠ operand2.
	<i>Arithmetic</i>	<i>Addition</i>	B	It adds two numeric values.
		<i>Division</i>	B	It divides one numeric value by another.
		<i>Modulus</i>	B	It returns the remainder of dividing one numeric value by another.
		<i>Multiplication</i>	B	It multiplies two numeric values.
<i>Subtraction</i>		B	It subtracts one numeric value from another.	
<i>Aggregation</i>	<i>Avg</i>	U	It returns the average of the values in an expression.	
	<i>Count</i>	U	It returns the number of the values in an expression.	
	<i>Max</i>	U	It returns the highest value in an expression.	
	<i>Min</i>	U	It returns the lowest value in an expression.	
	<i>Sum</i>	U	It adds the values in an expression.	

Table 5. Data windows of the event pattern metamodel.

Data window	Description
<i>Batching TimeInterval</i>	Tumbling window that batches events and releases them every specified time period (<i>years, months, weeks, days, hours, minutes, seconds, milliseconds</i>).
<i>Batching EventInterval</i>	Tumbling window up to the specified number of events (<i>size</i>).
<i>Sliding TimeInterval</i>	Sliding window by the specified time period (<i>years, months, weeks, days, hours, minutes, seconds, milliseconds</i>).
<i>Sliding EventInterval</i>	Sliding window by the specified number of events (<i>size</i>).

Table 6. Actions of the event pattern metamodel.

Data window	Description
<i>Email</i>	It specifies the email account(s) where the detected complex event will be sent. At least the mail sender (<i>from</i>), mail receiver (<i>to</i>), as well as the host (<i>host</i>), port (<i>port</i>), user name (<i>user</i>) and password (<i>password</i>) must be indicated. In addition, email subject (<i>subject</i>) and copy recipients (<i>cc</i>) may be indicated.
<i>Twitter</i>	It specifies the Twitter account where the detected situations (complex events) will be registered.

5.1.2. Restrictions for the event pattern metamodel

Next, restrictions for the event pattern metamodel are presented. Table 7 shows the validation rules that any event pattern model conformed to the metamodel must satisfy, as well as describing restrictions for each metaclass.

Table 7. Restrictions for the event pattern metamodel.






Metaclass	Restriction
<i>CEPEventPattern</i>	The event pattern name (<i>patternName</i>) must be specified.
	It must contain, at least, an event (<i>Event</i>) or data window (<i>DataWindow</i>).
	It must include the complex event (<i>ComplexEvent</i>).
	If a pattern operator (<i>PatternOperator</i>) is used, only one data window (<i>DataWindow</i>) containing all pattern conditions may be used.
<i>Link</i>	Two equal operators cannot be linked.
	Nested properties cannot be linked.
<i>ComplexEvent</i>	The complex event type (<i>typeName</i>) must be specified.
	It must contain, at least, a complex event property (<i>ComplexEventProperty</i>).
	Complex event properties must be unique.
<i>ComplexEventProperty</i>	The complex event property name (<i>name</i>) must be specified.
	It must be linked by an event property (<i>EventProperty</i>), an arithmetic operator (<i>ArithmeticOperator</i>) or aggregation operator (<i>AggregationOperator</i>).
<i>Event</i>	The event type (<i>typeName</i>) must be specified.
	It must contain, at least, an event property (<i>EventProperty</i>).
<i>EventProperty</i>	The property name (<i>name</i>) must be specified.
	It must be contained into an event (<i>Event</i>) or an event property (<i>EventProperty</i>).
	If it is a nested property, this property will not be able to contain some properties with the same name (<i>name</i>) for the same nested level.
	It cannot have inbound links with the same source.
<i>UnaryOperator</i>	It must have one inbound link.
<i>BinaryOperator</i>	It must have two inbound links.
	It must have inbound links with orders 1 and 2.
<i>NaryOperator</i>	It must have at least two inbound links.
	It must have inbound links with orders 1, 2...N, N being the total number of operators linked to the operator.
<i>AggregationOperator</i>	It must be linked to a complex event property (<i>ComplexEventProperty</i>).
<i>EveryDistinct</i>	The first inbound link must link an Event (<i>Event</i>) and the second one must link an event property (<i>EventProperty</i>).
<i>While</i>	The first inbound link must link a logical operator (<i>LogicalOperator</i>) or an <i>Every</i> , <i>EveryDistinct</i> or <i>FollowedBy</i> operator, and the second one must link a logical operator or comparison operator (<i>ComparisonOperator</i>).
<i>Range</i>	The <i>lowEndpoint</i> value must be less or equal than the <i>highEndpoint</i> value.
	It must link an <i>Until</i> operator with order 1.





































<i>Repeat</i>	The <i>count</i> value must be positive.
<i>TimeInterval</i>	It must have at least one attribute (<i>years, months, weeks, days, hours, minutes, seconds, milliseconds</i>) set to a positive value.
<i>TimeSchedule</i>	It must have at least one attribute (<i>dayOfWeek, dayOfMonth, month, hour, minute, second</i>) set to a positive value. The <i>dayOfWeek</i> attribute must be set to a value between 0 (Sunday) and 6 (Saturday), the <i>dayOfMonth</i> attribute must be set between 1 and 31, the <i>month</i> attribute between 1 and 12, <i>hour</i> between 0 and 23, <i>minute</i> between 0 and 59, and <i>second</i> between 0 and 59.
<i>WithinTimer</i>	It must have at least one attribute (<i>years, months, weeks, days, hours, minutes, seconds, milliseconds</i>) set to a positive value.
<i>DataWindow</i>	It must contain at least one event (<i>Event</i>).
<i>BatchingEventInterval</i>	It must have the <i>size</i> attribute set to a positive value.
<i>BatchingTimeInterval</i>	It must have at least one attribute (<i>years, months, weeks, days, hours, minutes, seconds, milliseconds</i>) set to a positive value.
<i>SlidingEventInterval</i>	It must have the <i>size</i> attribute set to a positive value.
<i>SlidingTimeInterval</i>	It must have at least one attribute (<i>years, months, weeks, days, hours, minutes, seconds, milliseconds</i>) set to a positive value.
<i>ArithmeticOperator</i>	The two operands must have a numeric type: <i>Integer, Long, Double</i> or <i>Float</i> .
<i>ComparisonOperator</i>	The two operands must have the same type: <i>Boolean, Integer, Long, Double, Float</i> or <i>String</i> .
<i>Value</i>	A value must be specified. If the value is <i>Boolean</i> , it must be <i>true</i> or <i>false</i> . It must be linked with a logical (<i>LogicalOperator</i>), comparison (<i>ComparisonOperator</i>) or arithmetic operator (<i>ArithmeticOperator</i>).
<i>Email</i>	The <i>to</i> attribute must have at least one correct email address. The <i>from</i> attribute must have at least one correct email address. If the <i>cc</i> attribute is used, it must have at least one correct email address. The <i>port</i> attribute must be greater or equal to 0, and less or equal to 65535. Common ports are: 25, 465, 475, 587 and 2525.

5.2. Concrete syntax

In addition to the definition of both the event pattern metamodel and restrictions, a graphical notation has been created for every element that can be used for designing an event pattern model. This concrete syntax for event pattern models is shown in Table 8.

Table 8. Concrete syntax for the event pattern metamodel.

Category	Name	Notation
	<i>Link</i>	
	<i>Value</i>	
<i>Simple Events</i>	<i>Event</i>	
	<i>EventProperty</i>	
<i>Complex Events</i>	<i>ComplexEvent</i>	

	<i>ComplexEventProperty</i>	
<i>Pattern Timers</i>	<i>TimerInterval</i>	
	<i>TimerSchedule</i>	
	<i>WithinTimer</i>	
<i>Pattern Operators</i>	<i>Every</i>	
	<i>EveryDistinct</i>	
	<i>FollowedBy</i>	
	<i>Range</i>	
	<i>Repeat</i>	
	<i>Until</i>	
	<i>While</i>	
<i>Logical Operators</i>	<i>And</i>	
	<i>Not</i>	
	<i>Or</i>	
<i>Comparison Operators</i>	<i>Equal</i>	
	<i>GreaterEqual</i>	
	<i>GreaterThan</i>	
	<i>LessEqual</i>	
	<i>LessThan</i>	
	<i>NotEqual</i>	
<i>Arithmetic Operators</i>	<i>Addition</i>	
	<i>Division</i>	
	<i>Modulus</i>	
	<i>Multiplication</i>	
	<i>Subtraction</i>	
<i>Aggregation Operators</i>	<i>Avg</i>	
	<i>Count</i>	
	<i>Max</i>	
	<i>Min</i>	
	<i>Sum</i>	
<i>Data Windows</i>	<i>BatchingEventInterval</i>	
	<i>BatchingTimeInterval</i>	
	<i>SlidingEventInterval</i>	
	<i>SlidingTimeInterval</i>	
<i>Actions</i>	<i>Email</i>	
	<i>Twitter</i>	

6. Evaluation and discussion

This section evaluates and discusses the novel DSMLs for the definition of CEP domains and event patterns described in Sections 4 and 5, respectively.

6.1 *ModeL4CEP evaluation*

As previously stated, we defined a model-driven approach for facilitating user-friendly design of complex event patterns in our previous work (Boubeta-Puig et al., 2014a), where a first version of a DSML for event pattern definition was proposed. However, this DSML, composed of an abstract syntax –an EPL metamodel along with its restrictions– and a concrete syntax establishing the relationship between the metamodel concepts and their graphical representation, offered several limitations compared to our new DSMLs proposed in this paper. First of all, a DSML for CEP domain definition was not considered in our previous work, lacking the mentioned advantages. Secondly, the graphical notation of the concrete syntax was less user-friendly. Thirdly, the EPL metamodel was incomplete and less understandable by end users as enumerated in the following lines: 1) actions to be carried out when detecting event patterns were not included; 2) events could not have nested properties; 3) the number of metamodel restrictions was lower, so the model validation process was less rigorous; 4) metaclasses lacked some properties providing useful information, such as the pattern creation date or domain name; 5) metaclass names were closer to EPL syntax than end users, such as “Output” instead of “EventPattern”, or “Length” data window instead of “SlidingEventInterval”; 6) an event pattern had to be modeled normally linking operators to operands, designing the pattern from right to left, instead of linking operands to operators with the purpose of designing the pattern in a more natural way from left to right; 7) there were some metamodel elements which did not provide valuable information for event pattern definition, for example, “PropertyReference” operand, since an event property could be referenced directly using a link.

To demonstrate the usefulness and strength of our novel DSMLs defined in this paper, we have done a comparative analysis (see Table 9) in which we determine how every metaclass of CEP domain and event pattern metamodels could be transformed into Esper EPL, Oracle EPL, StreamSQL and CCL code –some of the best known EPLs today–. Notice that other time units could be used for operators with time specified in seconds, by default they are indicated in seconds in the table. Next, the most significant aspects of this comparison are detailed.

Table 9. A comparison between the metaclasses of CEP domain and event pattern metamodels and their equivalent Esper EPL, Oracle EPL, StreamSQL and CCL code.

Metaclass	Esper EPL	Oracle EPL	StreamSQL	CCL
<i>EventPattern</i>	from pattern	MATCHING	FROM PATTERN	MATCHING
<i>Condition</i>	from where	FROM WHERE	FROM WHERE	FROM WHERE
<i>ComplexEvent</i>	insert into select	INSERT INTO SELECT	SELECT INTO	INSERT INTO SELECT
<i>ComplexEvent Property</i>	property as alias	property AS alias	property AS alias	property AS alias
<i>TimeInterval</i>	timer:interval (n seconds)	time_interval (n seconds)	interval(n)	n seconds
<i>TimeSchedule</i>	timer:at (* , * , * , * , * , *)		time(n)	AT n
<i>WithinTimer</i>	timer:within (n seconds)	WITHIN n SECONDS		
<i>Event</i>	Event	Event	Event	event
<i>EventProperty</i>	Property	Property	Property	property
<i>Value</i>	'string' Number true or false	'string' Number true or false	'string' Number TRUE or FALSE	'string' number TRUE or FALSE
<i>Every</i>	Every	EVERY		EVERY
<i>EveryDistinct</i>	every-distinct			DISTINCT ROWS
<i>FollowedBy</i>	->	FOLLOWED BY	-> and THEN	,
<i>Range</i>	[a:b]	BETWEEN a AND b	BETWEEN a AND b	BETWEEN a AND b
<i>Repeat</i>	[n]	BETWEEN 1 AND n	BETWEEN 1 AND n	n ROWS
<i>Until</i>	Until			UNTIL
<i>While</i>	While		FOREACH	FOR
<i>And</i>	and and ,	AND	AND and &&	AND and &&
<i>Or</i>	Or	OR	OR and	OR and
<i>Not</i>	Not	NOT	NOT and !	NOT and !
<i>Equal</i>	=	=	==	=
<i>GreaterEqual</i>	>=	>=	>=	>=
<i>GreaterThan</i>	>	>	>	>
<i>LessEqual</i>	<=	<=	<=	<=
<i>LessThan</i>	<	<	<	<
<i>NotEqual</i>	!=	!=	!=	!= and <>
<i>Addition</i>	+	+	+	+
<i>Division</i>	/	/	/	/
<i>Modulus</i>	%	%	%	mod
<i>Multiplication</i>	*	*	*	*
<i>Subtraction</i>	-	-	-	-
<i>Avg</i>	Avg	AVG	Avg	AVG
<i>Count</i>	Count	COUNT	Count	COUNT
<i>Max</i>	Max	MAX	Max	MAX
<i>Min</i>	Min	MIN	Min	MIN
<i>Sum</i>	Sum	SUM	Sum	SUM
<i>Batching TimeInterval</i>	win:time_batch (n seconds)	RETAIN BATCH OF n SECONDS	WITHIN n TIME	KEEP EVERY n SECONDS
<i>Batching</i>	win:length	RETAIN BATCH	WITHIN	KEEP EVERY

<i>EventInterval</i>	batch(n)	OF n EVENTS	n ON id	n ROWS
<i>Sliding TimeInterval</i>	win:time (n seconds)	RETAIN n SECONDS	SIZE n ADVANCE n TIME	KEEP n SECONDS
<i>Sliding EventInterval</i>	win:length (n)	RETAIN n EVENTS	SIZE n ADVANCE n TUPLES	KEEP n ROWS

Firstly, the *EventPatternCondition* metaclass represents the conditions to be fulfilled to detect a critical or relevant situation. This metaclass will be transformed into a search clause –normally used when neither pattern operator nor operand have been included in the conditions, for example, (from...where) in EPL Esper– or into a pattern clause –when conditions include some pattern operator or operand, for example, (from...pattern) in EPL Esper–.

The *ComplexEvent* and *ComplexEventProperty* metaclasses, which describe the complex event type to be created when detecting a pattern with their properties, are directly related to the clause that is responsible for creating the complex events of this type and insert them into a specific flow for them –insert into... select property as alias..., in Esper EPL–.

Regarding the pattern operands defined in the metamodel –*TimeInterval*, *TimeSchedule*, *WithinTimer* and *Event*–, Oracle EPL does not have any equivalent to *TimeSchedule*, and StreamSQL and CCL do not have equivalents to *WithinTimer*. In contrast, condition operands –*EventProperty* and *Value*– are similar for all these EPLs.

Regarding the pattern operators –*Every*, *EveryDistinct*, *FollowedBy*, *Range*, *Repeat*, *Until* and *While*–, there are more difficulties to find their analog clauses. The *Every* operator, which selects each event of the specified type, exists for all these EPLs, except for StreamSQL. Note that, although the latter does offer the *Every* operator, it can only be associated with a time interval and, therefore, does not provide the same functionality as the rest of languages –every event is selected by default in StreamSQL, not requiring the use of a specific operator for this purpose–.

Moreover, the *EveryDistinct* operator is only available for Esper EPL and CCL; however, a similar behavior to this operator could be defined using the *Every*, *And* and *Not* operators, such as `Every a = Event AND NOT b = Event (b.id = a.id)`.

Since Oracle EPL, CCL and StreamSQL make no distinction between *Range* and *Repeat* operators, the use of the BETWEEN . . . AND . . . operator can be proposed as a solution to obtain the same behavior; except to CCL that does offer n ROWS as a repetition operator.

Likewise, there are no exact matches for *Until* and *While*. On the one hand, the *Until* operator in CCL can only indicate a timestamp as stop condition, whereas the Esper *Until* operator is more generic, permitting the setup of other condition types. On the other hand, there is no operator identical to *While* of Esper EPL, but the FOREACH operator of StreamSQL and FOR of CCL can be considered similar to *While*.

The other metaclasses of this comparative –logical, comparison, arithmetic and aggregation operators as well as data windows– have equivalent elements for all the analyzed EPLs. Furthermore, some metaclasses have more than one equivalent operator for the same language as is the case, for example, of the *And* operator.

Therefore, event patterns can be defined as models by end users once and, thereafter, can be transformed into the concrete EPL provided by the CEP engine required at that moment. Obviously, this results in significant time saving and, especially, in minimizing the number of errors produced by programmers when writing the event pattern code by hand, since it is a fully automatic process.

6.2 A comparative study with other approaches for CEP domain and event pattern definition

We have conducted a comparative study of our DSML for CEP domains and event patterns (Model4CEP) with other existing approaches, detailed in Section 3. Table 10 summarizes the results of the study in which our solution is compared to the ontological approaches, whereas Table 11 shows the results in which our solution is compared to the model-driven ones, according to 16 criteria.

The results reveal that our novel solution has many strengths over other existing approaches. A noteworthy advantage is that our DSML for CEP domain definition facilitates the description of event types and properties for the domain for any user, expert in a particular domain but not in CEP. One of the most important contributions offered by this DSML is the unification of CEP domain description (event types and properties) by using models, hiding the implementation details necessary to define such

domains from domain experts. Regarding the other analyzed approaches, even though they support event types to be used for defining event patterns, most of them do not offer the possibility of describing a CEP domain composed of a set of event types together with their domain description. The lack of these elements in the domain will prevent the chance of sharing the domains among different modeling tools, which might belong to different users.

Another relevant strength is that our DSML for event pattern definition provides users with an intuitive and user-friendly way to describe both situations to be detected in a particular domain and the actions to be notified to interested users by email or social networking services, among others. One of the most important aspects of this DSML is the possibility of modeling patterns regardless of the implementation. Despite the fact that there are some proposals –(ALERT, 2013; Etzion & von Halle, 2013; Obwegger et al., 2011; Sen & Stojanovic, 2010)– that support event pattern graphical definition, all of them present limitations compared to our solution. Some approaches represent the event patterns as interconnected window nodes composed of buttons and dropdowns, which are used by users to add event properties together with their conditions, what may cause the design of vast and awkward event patterns. Other proposals use interconnected graphical nodes for event pattern definition where pattern conditions must be implemented in EPL code by hand; this is undoubtedly an impediment for non-expert CEP users.

Our approach, as well as some other included in Table 11, allows the creation of events with nested properties, providing greater flexibility when defining event types. Moreover, the possibility of defining hierarchies of events is really important, since it enables us to create complex events depending on other simple and complex events.

Another key aspect of our approach is its availability of numerous pattern timers, pattern operators, logical operators, comparison operators, arithmetic operators, aggregation operators and data windows. Although the majority of approaches include these types of operators and data windows, there are only a few available per type. As an example, Sen et al. (Sen & Stojanovic, 2010) only provide us with an aggregation operator (COUNT), data windows (WITHIN), logical operators (AND and OR) and temporal operator (SEQ). This is an important limitation when defining more complex event patterns.

Although most of approaches include actions to be executed when detecting event pattern conditions, these actions basically consist on generating response events in order to notify detected situations. Nevertheless, our approach supplies us with more sophisticated actions, such as email or social networking services.

Some related works deal with uncertainty in CEP; however Model4CEP does not. We consider it a relevant issue to improve our approach by providing it with the ability to model uncertainty in event patterns, for example, through Bayesian networks, as adopted by Cugola et al. (Cugola et al., 2015).

Table 10. Comparative study of Model4CEP with other existing ontological approaches.

Criteria	Model4CEP	ALERT13	Pas12	Pas14	Sen10	Stü09	Yao11
CEP domain definition	X						
CEP domain graphical definition	X						
Event pattern definition	X	X	X	X	X	X	X
Event pattern graphical definition	X	X			X		
Elements represented as graphical nodes	X	X			X		
Definition of events with nested properties	X						
Event hierarchy definition	X	X			X	X	X
Pattern timers	X	X	X	X	X	X	X
Pattern operators	X	X	X	X	X	X	X
Logical operators	X	X	X	X	X	X	X
Comparison operators	X	X	X	X	X	X	X

Arithmetic operators	X	X	X	X	X	X	X	X
Aggregation operators	X	X	X	X			X	X
Data windows	X	X	X	X	X	X	X	X
Actions	X	X	X	X	X	X	X	X
Uncertainty supported								

Table 11. Comparative study of Model4CEP with other existing model-driven approaches.

Criteria	Model4CEP	Bru14	Bru15	Cug10	Cug15	Etz13	Mul13	Obw11	Ter15a	Ter15b	Zan08
CEP domain definition	X	X	X			X					
CEP domain graphical definition	X					X					
Event pattern definition	X	X	X	X	X	X	X	X	X	X	X
Event pattern graphical definition	X					X		X			
Elements represented as graphical nodes	X							X			
Definition of events with nested properties	X	X	X					X		X	
Event hierarchy definition	X	X	X	X	X	X		X	X	X	X

Pattern timers	X	X	X	X	X	X	X	X	X	X	X	X
Pattern operators	X	X	X	X	X	X	X	X	X	X	X	X
Logical operators	X	X	X	X	X	X	X	X	X	X	X	X
Comparison operators	X	X	X	X	X	X	X	X	X	X	X	X
Arithmetic operators	X	X	X	X	X	X	X	X	X	X	X	X
Aggregation operators	X	X	X	X	X	X	X	X	X	X	X	X
Data windows	X	X	X	X	X	X	X	X	X	X	X	X
Actions	X		X			X		X	X	X	X	X
Uncertainty supported					X				X	X		

7. Conclusions and Future Work

In this paper, a graphical DSML has been proposed for CEP domain definition to facilitate any user, expert in a particular domain but not in CEP, the description of event types and properties for the domain. This DSML allows us to unify CEP domain descriptions by using models, hiding the implementation details necessary to define such domains from domain experts.

Besides, a highly expressive graphical DSML has been defined for event pattern definition. Its main purpose is to provide users with an intuitive and user-friendly way to describe both situations to be detected in a particular domain and the actions to be notified to interested users by email or social networking services, among others. The main advantage of this DSML is that business experts will easily define the pattern to be detected in the expert system in question even if they have no expertise on programming languages. Besides, this DSML allows modeling event pattern regardless of the language finally used for their implementation. Thanks to the use of MDD, every event pattern might be graphically designed once and then could be automatically transformed into any particular EPL, such as Esper EPL, Oracle EPL, StreamSQL or

CCL, as well as into any action code to be executed, such as XML. As a result, the patterns defined by the business expert can be detected and prioritized by the expert or intelligent business system in order to provide an appropriate response, consequently triggering a suitable action.

As part of our work-in-progress, we are creating two graphical modeling editors: an editor that supports the modeling of domains conformed to our CEP domain DSML and another editor supporting the modeling of event patterns conformed to our event pattern DSML.

The CEP domain editor will enable the graphical design and automatic validation of CEP domains, as well as exporting and importing them so that they can be shared and reused by other domain experts.

Regarding the event pattern editor, we already proposed one for designing event pattern models in our previous work (Boubeta-Puig et al., 2014a). However, these models conform to our previous event pattern DSML with the aforementioned limitations. In addition to solving these limitations, the key feature of our coming event pattern editor is its ability to reconfigure itself for different CEP domains, modeled by domain experts. The fact that the editor will be able to reconfigure the tool palette dynamically from different CEP domain models will allow users to enjoy a graphical interface adapted to the specific context required. Moreover, this graphical editor will enable CEP novices to concentrate on modeling both the situations to be detected and the actions to be carried out, hiding all implementation details from them. Thus, it will be possible to export and import the designed and validated pattern models as well as reusing them in different information systems through the transformation of these models into both the EPL code required by the chosen CEP engine and the code of actions to be carried out, thereby, bringing to reality the definition of critical or relevant situations in real time by non-technological users.

In our future work, we plan to extend Model4CEP with the ability to model uncertainty in event patterns, for example, through Bayesian networks.

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).

References

- ALERT. (2013). Active support and real-time coordination based on Event processing in FLOSS development. Retrieved February 17, 2015, from <http://www.alert-project.eu/>
- Anicic, D., & Fodor, P. (2014). ETALIS - Event-driven Transaction Logic Inference System. Retrieved June 12, 2014, from <https://code.google.com/p/etalis/>
- Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., ... Zettel, J. (2001). *Component-Based Product Line Engineering with UML* (1st ed.). London; New York: Addison-Wesley Professional.
- BEA. (2007). *Event Processing Market Pulse 2007*. ebizQ. Retrieved from http://complexevents.com/wp-content/uploads/2007/10/eventprocessing_survey.pdf
- Boubeta-Puig, J., Medina-Bulo, I., Ortiz, G., & Fuentes-Landi, G. (2012). Complex event processing applied to early maritime threat detection. In *Proceedings of the 2nd International Workshop on Adaptive Services for the Future Internet and 6th International Workshop on Web APIs and Service Mashups* (pp. 1–4). Bertinoro, Italy: ACM. <http://doi.org/http://doi.acm.org/10.1145/2377836.2377838>
- Boubeta-Puig, J., Ortiz, G., & Medina-Bulo, I. (2011). An Approach of Early Disease Detection using CEP and SOA. In *Proceedings of The Third International Conferences on Advanced Service Computing* (pp. 143–148). Rome, Italy: IARIA. Retrieved from http://www.thinkmind.org/index.php?view=article&articleid=service_computati_on_2011_6_30_10134
- Boubeta-Puig, J., Ortiz, G., & Medina-Bulo, I. (2014a). A Model-driven Approach for Facilitating User-friendly Design of Complex Event Patterns. *Expert Systems with Applications*, 41(2), 445–456. <http://doi.org/http://dx.doi.org/10.1016/j.eswa.2013.07.070>
- Boubeta-Puig, J., Ortiz, G., & Medina-Bulo, I. (2014b). Approaching the Internet of Things through Integrating SOA and Complex Event Processing. In Z. Sun & J. Yearwood (Eds.), *Handbook of Research on Demand-Driven Web Services: Theory, Technologies, and Applications* (pp. 304–323). IGI Global. Retrieved from <http://dx.doi.org/10.4018/978-1-4666-5884-4.ch014>
- Bruns, R., Dunkel, J., Lier, S., & Masbruch, H. (2014). DS-EPL: Domain-specific Event Processing Language. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems* (pp. 83–94). New York, NY, USA: ACM. <http://doi.org/10.1145/2611286.2611296>
- Bruns, R., Dunkel, J., Masbruch, H., & Stipkovic, S. (2015a). Intelligent M2M: Complex event processing for machine-to-machine communication. *Expert Systems with Applications*, 42(3), 1235–1246. <http://doi.org/10.1016/j.eswa.2014.09.005>
- Bruns, R., Dunkel, J., Masbruch, H., & Stipkovic, S. (2015b). Intelligent M2M: Complex event processing for machine-to-machine communication. *Expert*

- Systems with Applications*, 42(3), 1235–1246.
<http://doi.org/10.1016/j.eswa.2014.09.005>
- Chakravarthy, S., & Mishra, D. (1994). Snoop: an expressive event specification language for active databases. *Data & Knowledge Engineering*, 14(1), 1–26.
- Chandy, K. M., & Schulte, W. R. (2010). *Event Processing: Designing IT Systems for Agile Companies*. USA: McGraw-Hill.
- Chaudhuri, S., Dayal, U., & Narasayya, V. (2011). An overview of business intelligence technology. *Communications of the ACM*, 54(8), 88–98.
<http://doi.org/10.1145/1978542.1978562>
- Cugola, G., & Margara, A. (2010). TESLA: A Formally Defined Event Specification Language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems* (pp. 50–61). New York, NY, USA: ACM.
<http://doi.org/10.1145/1827418.1827427>
- Cugola, G., & Margara, A. (2012). Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*, 44(3), 1–62.
<http://doi.org/10.1145/2187671.2187677>
- Cugola, G., Margara, A., Matteucci, M., & Tamburrelli, G. (2015). Introducing uncertainty in complex event processing: model, implementation, and validation. *Computing*, 97(2), 103–144. <http://doi.org/10.1007/s00607-014-0404-y>
- Decker, G., Grosskopf, A., & Barros, A. (2007). A Graphical Notation for Modeling Complex Events in Business Processes. In *11th IEEE International Enterprise Distributed Object Computing Conference* (pp. 27–36). Annapolis, MD.
<http://doi.org/10.1109/EDOC.2007.41>
- Dunkel, J., Fernández, A., Ortiz, R., & Ossowski, S. (2011). Event-driven Architecture for Decision Support in Traffic Management Systems. *Expert Systems with Applications*, 38(6), 6530–6539. <http://doi.org/10.1016/j.eswa.2010.11.087>
- Eclipse Foundation. (2012). Emfatic. Retrieved April 5, 2014, from <http://www.eclipse.org/modeling/emft/emfatic/>
- EsperTech. (2015). Esper - Complex Event Processing. Retrieved February 28, 2015, from <http://www.espertech.com/esper/>
- Etzion, O., & Niblett, P. (2011). *Event Processing in Action*. Stamford, USA: Manning. Retrieved from <http://www.manning.com/etzion/>
- Etzion, O., & von Halle, B. (2013). *ER 2013 tutorial: modeling the event driven world*. Technology. Retrieved from <http://www.slideshare.net/opher.etzion/er-2013-tutorial-modeling-the-event-driven-world>
- Event Processing Technical Society. (2010). Event Processing Glossary - Version 2.0. Retrieved March 11, 2014, from http://www.complexevents.com/wp-content/uploads/2011/08/EPTS_Event_Processing_Glossary_v2.pdf
- Fowler, M., & Parsons, R. (2010). *Domain Specific Languages* (1st ed.). Massachusetts, USA: Addison Wesley.
- Gad, R., Boubeta-Puig, J., Kappes, M., & Medina-Bulo, I. (2012). Hierarchical Events for Efficient Distributed Network Analysis and Surveillance. In *Proceedings of the 2nd International Workshop on Adaptive Services for the Future Internet and 6th International Workshop on Web APIs and Service Mashups* (pp. 5–11). Bertinoro, Italy: ACM.
<http://doi.org/http://doi.acm.org/10.1145/2377836.2377839>
- García-Molina, J. (2013). Desarrollo dirigido por modelos: un nuevo paradigma de construcción de software. In J. García, F. Ó. García, V. Pelechano, A. Vallecillo, J. M. Vara, & C. Vicente-Chicote (Eds.), *Desarrollo de software dirigido por modelos: conceptos, métodos y herramientas* (pp. 289–306). Ra-Ma. Retrieved

- from <http://www.ra-ma.es/libros/DESARROLLO-DE-SOFTWARE-DIRIGIDO-POR-MODELOS-CONCEPTOS-METODOS-Y-HERRAMIENTAS/82019/978-84-9964-215-4>
- IBM. (2014, June 12). Operational Decision Manager [CT253]. Retrieved June 12, 2014, from <http://www-03.ibm.com/software/products/en/odm>
- JBoss Community. (2014). Drools Fusion. Retrieved November 13, 2013, from <http://www.jboss.org/drools/drools-fusion.html>
- Kleppe, A., Warmer, J., & Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise* (1st ed.). Boston: Addison-Wesley Professional.
- Luckham, D. (2002). *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. MA, USA: Addison-Wesley. Retrieved from <http://portal.acm.org/citation.cfm?id=515781>
- Luckham, D. (2012). *Event Processing for Business: Organizing the Real-Time Enterprise*. New Jersey, USA: Wiley. Retrieved from <http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470534850.html>
- Mulo, E., Zdun, U., & Dustdar, S. (2013). Domain-specific language for event-based compliance monitoring in process-driven SOAs. *Service Oriented Computing and Applications*, 7(1), 59–73. <http://doi.org/10.1007/s11761-012-0121-3>
- Obweger, H., Schiefer, J., Kepplinger, P., & Suntinger, M. (2010). Discovering Hierarchical Patterns in Event-Based Systems. In *IEEE International Conference on Services Computing (SCC)* (pp. 329–336). Miami, FL, USA. <http://doi.org/10.1109/SCC.2010.51>
- Obweger, H., Schiefer, J., Suntinger, M., & Kepplinger, P. (2011). Model-driven rule composition for event-based systems. *International Journal of Business Process Integration and Management*, 5(4), 344–357. <http://doi.org/10.1504/IJBPM.2011.043392>
- OMG. (2014). Unified Modeling Language. Retrieved May 7, 2014, from <http://www.uml.org/>
- Oracle. (2015). Oracle Event Processing. Retrieved February 3, 2015, from <http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html>
- Ortiz, G. (2007, April 12). Integrating Extra-Functional Properties in Model-Driven Web Service Development. Retrieved February 25, 2015, from <http://rodin.uca.es/xmlui/handle/10498/16036>
- Paschke, A. (2009). A Semantic Design Pattern Language for Complex Event Processing. In *Intelligent Event Processing, Papers from the 2009 AAI Spring Symposium* (pp. 54–60). Retrieved from <http://www.aaai.org/Library/Symposia/Spring/2009/ss09-05-010.php>
- Paschke, A. (2014). Reaction RuleML 1.0 for Rules, Events and Actions in Semantic Complex Event Processing. In A. Bikakis, P. Fodor, & D. Roman (Eds.), *Rules on the Web. From Theory to Applications* (pp. 1–21). Springer International Publishing. Retrieved from http://link.springer.com/chapter/10.1007/978-3-319-09870-8_1
- Paschke, A., Boley, H., Zhao, Z., Teymourian, K., & Athan, T. (2012). Reaction RuleML 1.0: Standardized Semantic Reaction Rules. In A. Bikakis & A. Giurca (Eds.), *Rules on the Web: Research and Applications* (Vol. 7438, pp. 100–119). Springer Berlin / Heidelberg. Retrieved from <http://www.springerlink.com/content/k13040qqqv262882/abstract/>

- Romero, D., Hermosillo, G., Taherkordi, A., Nzekwa, R., Rouvoy, R., & Eliassen, F. (2011). The DigiHome Service-Oriented Platform. *Software: Practice and Experience*, 43(10), 1205–1218. <http://doi.org/10.1002/spe.1125>
- Sen, S., & Stojanovic, N. (2010). GRUVe: A Methodology for Complex Event Pattern Life Cycle Management. In B. Pernici (Ed.), *Advanced Information Systems Engineering* (Vol. 6051, pp. 209–223). Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/978-3-642-13094-6_17
- Sen, S., Stojanovic, N., & Stojanovic, L. (2010). An approach for iterative event pattern recommendation. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems (DEBS)* (pp. 196–205). New York, NY, USA: ACM. <http://doi.org/10.1145/1827418.1827459>
- Software AG. (2014). Apama Analytics & Decisions Platform. Retrieved June 12, 2014, from http://www.softwareag.com/corporate/products/bigdata/apama_analytics/overview/
- Stahl, T., Voelter, M., & Czarnecki, K. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.
- Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2008). *EMF: Eclipse Modeling Framework* (2nd ed.). Addison-Wesley Professional.
- Stühmer, R., Anicic, D., Sen, S., Ma, J., Schmidt, K.-U., & Stojanovic, N. (2009). Lifting Events in RDF from Interactions with Annotated Web Pages. In A. Bernstein, D. R. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta, & K. Thirunarayan (Eds.), *The Semantic Web - ISWC 2009* (pp. 893–908). Springer Berlin Heidelberg. Retrieved from http://link.springer.com.diana.uca.es/chapter/10.1007/978-3-642-04930-9_56
- Sybase. (2015). SAP Sybase Event Stream Processor. Retrieved February 20, 2015, from <http://www.sap.com/uk/pc/tech/database/software/sybase-complex-event-processing/index.html>
- Terroso-Sáenz, F., Valdés-Vela, M., Campuzano, F., Botia, J. A., & Skarmeta-Gómez, A. F. (2015). A complex event processing approach to perceive the vehicular context. *Information Fusion*, 21, 187–209. <http://doi.org/10.1016/j.inffus.2012.08.008>
- Terroso-Saenz, F., Valdes-Vela, M., den Breejen, E., Hanckmann, P., Dekker, R., & Skarmeta-Gomez, A. F. (2015). CEP-traj: An event-based solution to process trajectory data. *Information Systems*, 52, 34–54. <http://doi.org/10.1016/j.is.2015.03.005>
- TIBCO. (2015). StreamBase Studio. Retrieved February 15, 2015, from <http://www.streambase.com/products/streambasecep/streambase-studio/>
- Uhm, Y., Lee, M., Hwang, Z., Kim, Y., & Park, S. (2011). A multi-resolution agent for service-oriented situations in ubiquitous domains. *Expert Systems with Applications*, 38(10), 13291–13300. <http://doi.org/10.1016/j.eswa.2011.04.150>
- Vincent, P. (2010, March 12). The Return of the Expert System? Retrieved from <http://www.tibco.com/blog/2010/03/12/the-return-of-the-expert-system/>
- W3C. (2014a). OWL Web Ontology Language. Retrieved June 9, 2014, from <http://www.w3.org/TR/owl-features/>
- W3C. (2014b). RDF Schema 1.1. Retrieved June 5, 2014, from <http://www.w3.org/TR/rdf-schema/>
- Yao, W., Chu, C.-H., & Li, Z. (2011). Leveraging complex event processing for smart hospitals using RFID. *Journal of Network and Computer Applications*, 34(3), 799–810. <http://doi.org/10.1016/j.jnca.2010.04.020>

- Yuan, S.-T., & Lu, M.-R. (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), 3671–3694. <http://doi.org/10.1016/j.eswa.2008.02.024>
- Zang, C., Fan, Y., & Liu, R. (2008). Architecture, Implementation and Application of Complex Event Processing in Enterprise Information Systems Based on RFID. *Information Systems Frontiers*, 10(5), 543–553. <http://doi.org/10.1007/s10796-008-9109-0>