Event-driven Reactivity: A Survey and Requirements Analysis

Kay-Uwe Schmidt[1], Darko Anicic[2], and Roland Stühmer[1]

[1] SAP AG, Research, Vincenz-Prießnitz-Straße 1, 76131 Karlsruhe
˜http://www.sap.com
[2] FZI Forschungszentrum Informatik, Haid-und-Neu-Straße 10-14, 76131 Karlsruhe
˜http://www.fzi.de

Abstract. Despite the huge popularity of event processing nowadays, there is a big gap between the potential usefulness of event-driven processing and the current state of the practice. One of the main reasons is the lack of a comprehensive conceptual model for the event-triggered reactivity and the corresponding framework for its management. In this paper we survey the current state of the art in event-driven architecture with special focus on event and action processing. We describe the prerequisites of a completely novel conceptual model for describing reactivity that is more close to the way people react on events: based on the ability to identify the context during which active behavior is relevant and the situations in which it is required. This approach opens a completely new view on the event processing as the way for managing a very valuable knowledge asset of every enterprise - knowledge how to react (make decisions) in event-driven situations.

1 Introduction

Event-driven processing becomes ever important in various application domains, ranging from traditional business applications, like supply-chain management, to the entertainment industry, like on-line gaming applications. The market value should increase tenfold by 2010 and should reach something like $4bn in total (source: IBM). The most relevant market research companies, like Gartner or Forrester[3] predict the key role of even-driven processing for making business more agile. Indeed, the main benefit of "eventizing" business systems is that event processing introduces a kind of reactive dynamics in the system, that enables active responding on signals sensed/derived from the context (internal, external), which the system is functioning in. Obviously, such kind of reactivity, which we call event-triggered reactivity (EtR), opens great opportunities for system/process improvements.

However, despite considerable research efforts put in this domain, the current development is just a top of the iceberg regarding the theoretical usefulness of

[3] cf. Forrester research: "CEP (complex event processing) Adoption Is Broader, Deeper, And More Business-Driven Than IT May Expect", January 31, 2008

event processing. One can find many causes for this "problem" (like lack of usable tools, editors, standards, ...) but in the nutshell of the problem is a kind of the deficiency in dealing with the ad-hoc nature of events, i.e. unpredictable (but controlled) appearance of events. Unpredictability means that, for example, we don't know in advance when an event will be issued, but we know that such an event can appear. In that sense, we usually don't know exactly, in advance how we will react on an event, but we know that we will react. Indeed, we design an event so that it triggers reaction, but we react dependently on other circumstances (e.g. whether a server went down during working hours or not). In fact, an event triggers reaction and these "other circumstances" determine the reaction. More interestingly, we can react on the same event in different ways, since these "other circumstances" are different.

Therefore, the main problem in current approaches is that the level of abstraction used in representing events and conditions is too low. It requires too much puzzling (of events) without having any more abstract commonality between events (analogy is using only the form of puzzles as the feature to find the next suitable puzzle: really, it is a hard work if all puzzles are black and every puzzle has unique form). If the events we are dealing with are more complex, the possibility/easiness to build useful applications is higher. On the other side, people are processing events in another (more abstract) way: they are reacting on *situations* according to the *context*. A central issue in our reactivity (and in general in reactive and proactive systems) is the ability to identify the context during which active behavior is relevant and the situations in which it is required. The concept of situation is an extension of the concept of composite event in its context awareness capability; it results in additional expressive power, flexibility and usability. The main point is that puzzling events (and actions) in complex structures can be done now by using this abstract description as an additional feature, which certainly alleviates the whole process.

In this paper we provide a survey and requirements analysis for handling the event triggered reactivity. We distinguish the analysis between a non-logic and logic-based view on handling the event triggered reactivity. The non-logic view does not consider formal (logical) representation of elements in reactive rules. On the other hand, the logic-based approach follow the line of thinking that semantics of complex relationship in both a single reactive rule and ruleset should be described formally. In this way, a *control* mechanism for an automated execution in reactive systems is established by means of logic. Furthermore we distinguish the analysis between event processing (i.e., change detection) and action processing (i.e., reaction on change).

Sections 2 and 3 recap the state of the art (for event and action processing, respectively) in event-triggered reactivity. Section 4 derives requirements for handling Event-triggered Reactivity. Section 5 gives an outlook of our future plans.

2 Event Processing

Complex Event Processing (CEP) and Event Stream Processing (ESP) Complex Event Processing and Event Stream Processing (ESP) are two fields of research concerned with the handling of events. Traditionally the fields tried to solve different problems in event processing using different approaches. The next paragraphs will outline these approaches, as they are seen e.g. by David Luckham in [1] and by others.

Events may happen in a stream or in several streams, a cloud. The field of ESP is concerned with extraction of events from a stream. Thus ESP handles events that are totally ordered by time. Further emphasis of ESP lies on efficiency for high throughput and low latency. Processing is done by analyzing the data of the events and selecting appropriate occurrences. Long-running queries produce results regularly; an analogy may be drawn to signal processing.

CEP, on the other hand, is more focused on complex patterns of events. To detect these patterns CEP takes more time and memory than ESP. CEP is concerned with clouds of events, which yield only a partial temporal order of events. Other partial orders of interest for CEP are causality, association, taxonomy, ontology. Rather than to signal processing, an analogy may be drawn to higher level situational inferencing, in comparison to ESP.

However, because CEP and ESP nowadays adopt each others approaches, the two worlds become mingled and sources as [2] declare them one and the same field.

2.1 Non-logic-based Approaches

Early event specification languages were developed for active databases [3]. They use complex event specifications to facilitate database triggers which do not only listen to simple events but observe complex combinations of events until the trigger procedures are executed.

Simple events carry a type, their occurrence time and possibly other parameters that can be used in data analysis to help in detecting event patterns or be part of a computation after detection. Building on the event types one can create complex nested expressions, using operators like And, Or, Sequence, and others that have been proposed since the start of CEP in the 1990s. Complex event specifications are patterns of events which are matched against the streams of events that occur during the run-time of the system. These patterns consist of simple event types and event operators. Simple events are the basic events the system can detect. Complex events are detected from occurrences of one or more of them. All simple events have a simple event type, which for a database application might be insert, update and delete. The types are used as placeholder in event patterns.

Event patterns are structured by event operators. A given operator might have several event types as arguments and e.g. stipulate that the constituent events must occur in sequence. An event detector for the given pattern functions as a stream pattern matcher and listens for events that satisfy the type

constraints and together satisfy the semantics of the given operator, e.g. occurred in sequence. Many operators were proposed in the past and the following paragraphs discuss several event pattern languages and their operators. Usual operators offered by many languages include disjunction, sequence and accumulation.

One early active database system is HiPAC [4]. It is an object-oriented database with transaction support. HiPAC can detect events only within a single transaction. Global event detectors are proposed which detect complex events across transaction boundaries and over longer intervals, but no further details are given. Ode [5] is another active database system with a language for the specification of event expressions. The language is also referred to as Compose. Ode proposes several basic event operators and a large amount of derived operators for ease of use and shorter syntax. The last of the classical event specification languages discussed here is Snoop [6] and its successor SnoopIB. Snoop provides the well known operators And, Or, as well as Sequence. The remaining operators are: Not, Any, A, A*, P, P* and Plus. Early work on Snoop views events as having instantaneous occurrences. This also holds true if an event is complex and its constituents span an interval of time. As a result the time of detection is used for the occurrence, instead of the interval from the start of the first constituent event to the end of the last constituent event. Consideration of only the time of detection is termed detection-based semantics. It poses problems with nested sequences as pointed out in [7]. Interval-based semantics for Snoop is called SnoopIB and was first published in 2006 [8].

Selection and consumption of events define which occurrences participate in a complex event. Both terms are an integral part of the semantics of an event definition. Selection defines the choice of events if there are more than one event of a required type that have not yet been consumed. Consumption is concerned with the deletion of events when they cannot be part of further complex events.

Other approaches to event pattern languages include statements reminiscent of SQL. Two examples are StreamSQL[4] and Continuous Computation Language CCL[5]. Queries in these languages match patterns in streams instead of database tables. Queries are long-running and produce incremental results in contrast to SQL queries. In CCL sliding windows are supported, joins are possible to form complex events and patterns may be specified using the operators conjunction, disjunction, sequence and negation. All operators can only be applied to bounded windows of events. Complex events have to adhere to SQL schemata which prohibits nested sets, for example an events that includes a previously unknown number of constituents. Although the well known syntax of SQL might help with the adoption of these languages, a seamless integration of an action part seems hard to accomplish. None of these approaches so far interact with the business vocabulary. They also do not consider the context of events and the relationship between events and actions.

[4] http://streambase.com/developers/docs/latest/streamsql/index.html
[5] http://www.coral8.com/system/files/assets/pdf/5.2.0/Coral8CclReference.pdf

Many of the aforementioned event languages belong to their respective database management system, or prototype thereof. Three of them are described here, which have noteworthy implementation details: The Ode approach conducts complex event detection by using automata. SAMOS uses colored Petri nets. Sentinel uses a graph based approach.

Complex event detection in Ode [9] is implemented using automata. Input for the automata is a stream of simple events. Ode thus transforms complex event expressions into deterministic finite automata. For sub-expressions which are complex events themselves, the process is done recursively. Atomic simple events are ultimately represented as automata of three states; a start state, an accepting state, entered upon detection of the simple event occurrence, and a non-accepting state, entered upon detection of any other simple event. Apart from providing the implementation, automata are a convenient model to define semantics of complex event operators. A downside of automata is that an automaton cannot accept overlapping occurrences of the same complex event. Also event parameters pose a problem. They are either stored outside of the automaton, or the automaton is increased greatly in the number of states to accommodate the different parameters and possible values thereof.

Complex event detection in SAMOS [10] is implemented using Petri nets. Each primitive event type is represented by a Petri net place. Primitive event occurrences are entered as individual tokens into the network. Complex event expressions are transformed into places and transitions. Where constituent events are part of several expressions, duplicating transitions are used to connect the simple event with the networks requiring it. This results in a combined Petri net for the set of all event expressions. Petri nets, like automata provide a model of the semantics of event operators. Also the detection of overlapping occurrences is possible. Event parameters are stored in tokens and flow through the network. Although the tokens are individual, there is no mechanism to deterministically choose a token if there are more than one in a single place.

Sentinel [11] is an active object-oriented database implementing complex event detection for the Snoop operators. Event detection follows a graph based approach. The graph is constructed from the event expressions. Complex expressions are represented by nodes with links to the nodes of their subexpressions, down to nodes of simple events. Event occurrences enter the bottom nodes and flow upwards through the graph, being joined into composite occurrences. The graph is a directed acyclic graph and generally does not form a tree for two reasons: nodes may have several parents, when their represented expression is part of more than one complex events, and secondly there is no single root node, when there is no overarching, single most complex event. A possibly conceived drawback of Snoop compared to the previously mentioned implementations is that the data structures of Snoop do not represent and even clarify the semantics of the event expressions. The logic of Snoop is hidden in the implementation of each graph node. However the semantics of Snoop is defined externally, using event histories and describing the operators as mappings from simple event histories to complex event histories. Furthermore Snoop defines the selection and

consumption of simple events for the concurrent detection of overlapping complex events. The four alternative definitions, Recent, Chronicle, Continuous and Cumulative context were described earlier.

2.2 Logic-based Approaches

In order to capture relevant changes in a system and respond to those changes adequately, a number of logic-based reactive frameworks have been proposed. It is a challenge to ensure that a reactive system handles changes (events) properly and in an automated manner. By handling changes it is meant that a reactive system undertakes certain actions (reactions) which can be seen as an act of change propagation. In general, an action changes the state of the system or triggers a new event (more details about actions are given in Section 3, particularly Section 3.2). Therefore it is a question how to effectively control the whole reactive system, i.e. how machines can keep executing certain activities, ensuring at the same time the system consistency. In order to achieve this goal, logic-based reactive approaches combine a reactive system (e.g., Event-Condition-Action system) with deductive capabilities of a particular logic.

IBM has been developed an event processing tool available in IBM Tivoli Enterprise Console. The engine is capable to processes complex events and executes Event-Condition-Action rules. The Prolog programming language [12] is used as an underlying formalism for specifying events, event filters, and actions.

A Logic Programming (LP) routed approach for dealing with events has also been proposed in [13]. More precisely, deductive rules are utilised for creating implicit events. Motivation to use datalog-like rules, extended with stratified negation was justified there by a number of reasons. First, rules serve as an abstraction mechanism and offer a higher-level event description. Rules allow for an easy extraction of different views of the same reactive system. Rules are suitable to mediate between the same events differently represented in various interacting reactive systems. Finally, rules can be used for reasoning about causal relationship between events.

In [14] a homogeneous reaction rule language was proposed. This approach combines complex event and action processing, formalisation of reaction rules in combination with other rule types such as derivation rules, integrity constraints, and transactional knowledge updates. Motivation to use logic in reactive systems is also justified there with the need to correctly and effectively process the event-based behavioural and semantics of reaction rules. However one drawback of this approach is the query based complex event processing (the same issue holds for [13]). In order to simulate active behaviour of ECA systems with passive Logic Programming based systems, the complex event processing in above mentioned approaches, is realised by frequently issued queries. For instance, a complex event pattern is encoded as a query and issued periodically. If such a query retrieves an answer, the system identifies that as an occurrence of a corresponding complex event. The complex event is than used for selecting ECA rules that need to be executed. The query based complex event processing realised with passive LP systems are not capable to identify complex events as soon as they emerge, but

at the time when a corresponding query is processed. This issue may have certain implications with respect to the intended semantics of complex ECA rules.

3 Action Processing

Although event processing is a major part in event-triggered reactivity there is more to it: actions. Action processing is the task of executing actions triggered by events in well-defined contexts and situations. This section deals with non-logic and logic-based approaches of triggering actions caused by events.

3.1 Non-logic Based Approaches

Many active database systems not only specify an event language but also a reaction rule language. Reaction rules such as Event-Condition-Action (ECA) rules complement the event specification with a condition to be evaluated upon event occurrence and a corresponding action to take. The term ECA rule was first used in conjunction with the HiPAC active database [15]. ECA rules are a generalization of several methods to achieve active behavior, such as triggers and production rules, which had been in prior existence but treated separately. The ECA rule approach divides the rule execution in three parts, event, condition and action handling. The parts are processed concurrently but work with different input. Event processing deals with transient input, i.e. the events. Although events are objects, they are usually not made persistent but are used in an online, real-time fashion to deduce complex observations and thereby consume the less-refined input. The output from the event detection is knowledge about complex distributed incidents happening in a system. This knowledge incorporates the information from individual events in an accumulated fashion. Conditions, on the other hand, deal with persistent data or knowledge. They may be viewed as queries to a database or a knowledge base. Although the outcome of conditions may change over time, the condition evaluation generally deals with long-lived data, which unlike events may not be time-dependent.

Many reaction rule languages have been proposed in the past. Not all obey the separation of event, condition and action specifications. Computational issues have been identified early. Large amounts of memory and computational effort may be wasted on the detection of events which are subsequently discarded when corresponding conditions are not met. This led many language designers to dilute the event and condition parts. Contrary to a declarative approach it is then up to the rule author to revise events expressions for run-time efficiency.

Ode [5] for example, promotes so-called EA rules, instead of ECA rules, where the conditions are folded into the event specification. This is done by mixing event expressions with filters. These filter predicates can impose conditions on events in order to discard occurrences early. Thereby the deleted occurrences do not use further resources or become part of complex events which will not be needed.

Similarly the logical language ADL [16] has (EC)*A-rules. Like the event masks in Ode the language tries to achieve finer granularity by mixing event and condition specifications.

Production systems execute rules also termed productions based on the evaluation of some conditions. The most often used algorithm in production systems is RETE. RETE is designed by Charles Forgy and described in [17]. It is a forward-chaining algorithm for evaluating production rules (PRs)[6]. Production rules are traditionally used in conjunction with a working memory. The working memory is a potentially large set of objects or values or objects which may be referenced in the conditions of rules. Actions are fired when a condition becomes true.

However, conditions can be expensive to evaluate; a large working memory requires a lot of elements to be taken into consideration, and complex conditions require nested checks to be performed. Once a single element in the working memory is added, changed or deleted, the conditions of many rules may change their outcome. In the worst case every rule must be reevaluated. The RETE algorithm remedies this situation by introducing state-saving of the evaluation process between changes to the working memory. The condition evaluation needs not to be fully recomputed. This is accomplished by dividing the conditions into a hierarchical network of nodes, each doing a single comparison, filter operation, join, etc. Every node has a so-called memory which stores the objects that fulfill the constraints of its node. When an object is changed, the network does not need to be completely refilled, but only the changed object is re-fed into or removed from the network. A classic RETE network is divided into two parts; the first, called the alpha network, contains nodes with one input edge. These nodes perform filter operations using single conditions or constraints. The second part, the beta network, contains nodes with two input edges, joining objects from two subordinate nodes. A third type of node exists that models the top nodes. The rule actions are attached to these nodes.

The Rete algorithm has two successors: Rete II and III but they have not been published. There are other optimized algorithms based on Rete, TREAT [18] and LEAPS [19] being two examples. Variations on Rete are implemented in many current rule engines.

3.2 Logic Based Approaches

The action part in reaction rules may additionally be described in a formal way (using a particular logic). The formal description, does not help only in controlling the execution of actions, but also allow for reasoning over complex actions[7].

A homogeneous reaction rule language [14] is a language for extended ECA rules (not only for CEP) implemented in a logic framework. Moreover the lan-

[6] Production rule (PR) is also called a condition-action (CA) rule.

[7] Complex actions in general case may implement non-trivial procedures such as business workflows.

guage is extended with Reaction RuleML, as a platform independent rule interchange format, and rule serialization in XML. The specifications of event, condition and action are strictly separated. More precisely rules are expressed as tuples (T, E, C, A, P, EL), consisting of T time, E event, C condition, A action, P post condition and EL contingency action, "else". Parts might be left blank, i.e. always satisfied, stated with "_", e.g., (T, E, _, A, _, _). Blank parts might be completely omitted, leading to specific types of rules, e.g. standard ECA rules: (E, C, A), or extended ECA rules with post conditions ECAP: (E, C, A, P). Event specifications of the homogeneous reaction rule language follow the Snoop [20] operators, redefined with an interval-based semantics. Different consumption policies are mentioned but do not seem to adhere to the Snoop consumption modes Recent, Chronicle, Continuous and Cumulative.

The need to declaratively describe the action component in an ECA rule has also been recognised in [21]. Particularly Calculus of Communicating Systems (CCS) [22] was chosen to formally specify complex actions, and reason about their behavioural aspects. As a part of the Process Algebra family, CCS is suitable for the high-level description of interactions, communications, and synchronizations between a collection of concurrent processes, and hence an appropriate mechanism for reactive systems in general. Although CCS is by no means powerful formalism, its use in practical world is still limited. There remains a huge gap between the model and the code, i.e., between the specification of desired behaviour and the program that implements it [23].

4 Requirements for Handling Event-triggered Reactivity

Based on the survey of state-of-the-art in event-triggered Reactivity we derived several challenges as requirements for future systems leveraging the full potential of event-triggered reactivity.

4.1 Non-logic-based Requirements

Table 1 shows a summary of todays open issues in non-logic-based event processing. Furthermore, Rete was designed for evaluating production rules (CA rules) only. Considering ECA-like rules, the event detection also needs to be considered. ECA rules are triples of event, condition and action specifications. The rules are fired, when the corresponding event has occurred, only if the condition is fulfilled. The event may be a complex event specification. Instead of managing events and conditions separately, the two should be integrated into the Rete network, extending it for temporal data and therefore event processing. This has been proposed in [24] but no results have been published.

Events are manifested in first class objects. Objects provide a straight-forward facility to store parameters and other data about the event, and propagate them through the detection process. To process events, Rete could be extended with new nodes that handle events instead of working memory elements. Events may be filtered and joined according to event specifications (similarly as working

Table 1. Open issues in non-logic-based event processing

Technology	Open Issues	References
HiPAC	- detection restricted to a transaction - only detection-based semantics - no notion of contexts and situations	[4]
Ode, Compose	- non-declarative mix of event and condition - automata do not detect simultaneous complex events of one type - automata cannot easily handle event parameters - only detection-based semantics - no notion of contexts and situations	[5]
SAMOS	- no clear strategy for event consumption - only detection-based semantics - no notion of contexts and situations	[10]
Snoop, SnoopIB	- missing integration of actions - no notion of contexts and situations	[20, 11, 8]
StreamSQL,	- not extensible for actions, ECA	
CCL	- detection only within (predefined) windows - no notion of contexts and situations	
ADL	- non-declarative mix of event and condition - no notion of contexts and situations	[16]

memory elements are filtered and joined according to the condition specification). Complex events and complex conditions may be joined to fire rule actions on activation of the join node.

The integration has several advantages. No separate data structures are needed for graph based event detection and the Rete network. Also computation cycles and memory is saved by consuming events earlier, when corresponding conditions are not fulfilled: Conditions have to be positively evaluated during the detection interval of an event. Therefore when a condition is not satisfied, no constituent events need to be collected for that event. Such an implementation retains the separation of event, condition and action in a purely declarative language but benefits from a resource-saving internal execution.

Mixing events and conditions on the language level may be supported, if needed. This allows for events that are masked by a condition, such as the mask construct in the ODE language [9] or (EC)*A rules in the ADL language [16] (see Table 1). Instead of joining highly complex events and conditions this is done at arbitrary levels of the network. Such masked events do not occur while their associated condition expression is not fulfilled. Furthermore, mixing event nodes and condition nodes enables conditions to see constituent event occurrences and

include them in their evaluation, i.e. join operation. This way the condition part of a rule can be dependent on the occurrences detected in the event part.

4.2 Logic-based Requirements

Event-condition-action and production rules are considered as an appropriate form of rules for programming systems which need to detect changes and respond on them automatically. However in general case, their use may be very unpredictable with respect to their intended semantics [25]. In general case, execution of an event may trigger other events, and these events may trigger even more events. There is neither guarantee that, such a chain of events will terminate, nor that states (through which a reactive system passes) are valid, i.e. *termination* problem. Further on, two reactive rules with the same execution priority may lead the system to two different states of the whole system. The system cannot be in two states at the same time. Therefore a rule base needs to be *confluent* (i.e., two rules triggered in an initial state lead the system, not to two, but to a single final state, regardless of the order which any subsequent simultaneously triggered rules are selected for firing). The next issue is the rule *ordering* (i.e., two rules may produce different effects if the first rule is scheduled before the second and vice versa). Termination, confluence, ordering, and similar issues have been recognised and extensively discussed in the area of Active Databases [3]. Currently logic-based approaches lack a comprehensive framework capable to deal with these issues.

Apart from classical concepts in event-triggered reactive systems (i.e., events, conditions and actions), those systems should also consider new notions - *situations* and *contexts*. In many cases there is a gap between current reaction rules that enable reaction to an (atomic or complex) event, and the reality, in which reaction is relevant only in a certain context in response to patterns over reaction histories [26]. As event-driven reactive systems act autonomously, a central issue is the ability to identify that context during which active behaviour is relevant and the situation in which it is required.

One of advantages of logic-based approaches for handling EtR is (automated) *reasoning* service. However this very important feature is limited on reasoning over one component of reactive rules, i.e. either on events [13], conditions, or only on actions [21]). An *appropriate formalism* for expressing all components of reactive rules (i.e., events, conditions, actions, but also situations and contexts) is missing. Such a formalism would allow for reasoning over events, condition and actions uniformly as well as discovering new relationships between these constructs w.r.t a given situation and context. Further on, reactive systems are state-changing systems. Hence instead of a logic that may be used only for reasoning in one particular state, appropriate EtR processing necessitates a logic for *state-changing reasoning*. The purpose of such a logic is to control state-changing action execution, keeping a reactive system always in a consistent state. By executing reactive rules, the system changes its states. In this transition, every state in which the system enters, needs to be a legal state. However if the inference engine, searching for a possible execution path, enters to an illegal state

such a state-transition should be rolled back. In this way, automated execution of reactive systems should be also controlled in an automated manner. In this respect an appropriate logic should be used in implementation of such advanced reactive systems.

4.3 Vision: Event-triggered Reactivity

We envision an holistic approach: Event-driven Reactivity. The unique handling of the different constituents of an event-driven architecture namely events, actions, conditions, contexts and situations will support the realization of next generation efficient and manageable event-driven (reactive) applications. It will firstly decrease the complexity of setting-up/evolving event-driven applications, that nowadays requires lots of manual work, especially in defining what an event is; secondly increase the benefits (added value) of such applications, which are currently constrained on the complex monitoring of events; and thirdly open new possibilities to apply them in highly dynamic and distributed environments. This vision will be realized through the following set of requirements:

- Efficient modeling of the sense-and-respond (reactive) nature of a system, especially its contextualization
- Comprehensive management of the reactivity life cycle of a system, including automatic discovery of relevant situations, efficient detection of events and reasoning about actions
- Efficient implementation of the reactivity life cycle management

In fact, we argue that the ECA (event-condition-action, such as it is) model is too simple presentation of the (intelligent) event processing nature which results in the already mentioned insufficiencies of event processing applications. Additionally we argue that the role of an efficient *context detection* process is inevitable for the efficient event processing and is totally neglected in the literature. Consequently, we argue that a unified mechanism for formal representation of all phases in the reaction cycle is needed for efficient and complex event processing [27]. In fact, we can go a step further and say that by using a richer conceptual model for describing reactions on events, we are not any more talking about simple processing of events, but rather about the management of a very valuable knowledge asset of every enterprise (system), i.e. knowledge howto react (make decisions) in event-driven situations[8].

[8] Note that traditional ECA rules are not very suitable for the description of knowledge assets since they describe only a part of relevant knowledge. Indeed, if we consider a rule "ON three servers clashes within one hour IF during workhours DO rescue", then it codes just the "basic" knowledge, or the simplest knowledge related to the particular situation. Much more interesting would be rules with more details, like "ON three servers clashes within one hour IF during workhours and already tried methods are rescue#1 before rescue#2 DO rescue#3", which can be provided by our model. Additionally, the rules are too coarse grained to be efficiently reused.

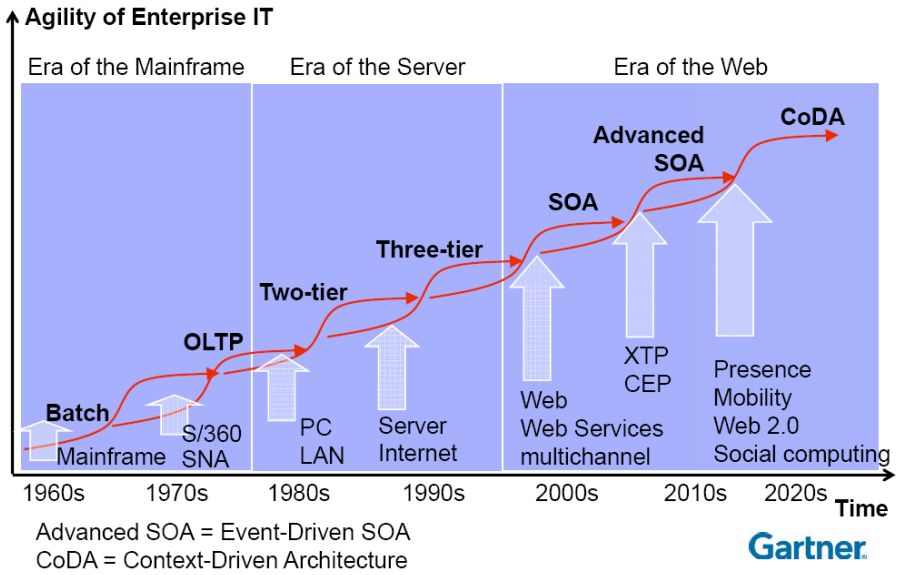

Fig. 1. Gartner view on context: The Next Step for Software and Communications Architectures.

Note that having context as a first class citizen in the event processing is currently completely missing in the literature for event processing. Nevertheless, context-based event processing is the next "big thing" and should shape the future of the computing, as given in the recently issued Gartners visionary view, depicted in Figure 1. The graphics estimates Context-Driven Architecture (CoDA) as the most promising paradigm that will extend SOA.

Reasoning about situations and context opens new possibilities for event-triggered reactivity. If we would have situations as formal logic models, then some very interesting reasoning services can support the whole event processing. For example, we can define two situations as conflicting to each other and try to avoid running the whole system in such a state. The system can check formally the consistency of the system and backtrack if a conflict (meaning inconsistency in the system) is to appear. It can help us to optimize reactions on situations. Another service would be the synchronization of situations if we consider that two or more reactions will run in parallel, which is a quite natural assumption in the rich-event systems.

5 Future Work

In the future we intend to develop a new conceptual model and architecture of event-triggered reactivity (EtR) that will resolve drawbacks of existing approaches for modeling reactivity, by introducing novel concept (situation and context) and its formal, logic-based representation. Moreover we will develop a model for managing the whole life cycle of EtR, including: a) Language for

modeling EtR concepts (e.g. situations, context), user-friendly editor based on pattern modeling metaphor, as well as methods for ensuring the consistency[9] of such a rule base and its interoperability with other reactive systems; b) New methods and tools for the automatic discovery of complex event and situation patterns from stream data by taking into account their evolution as well; c) New algorithms for scalable ECA reasoning, based on the selected logic and its implementation in a new reasoning engine that will serve as the event-, condition- and action-handler in a reactive system. Furthermore we plan to realize, test and refine an integrated software framework for the management of EtRs life cycle, containing elements of the distributed event processing, that can be easily deployed in the selected legacy landscapes. Finally we aim the Development of use cases, their implementation, testing and evaluation in real-world pilot studies in order to validate proposed model and framework.

References

[1] David C. Luckham. Whats the difference between esp and cep? Online Article, August 2006.

[2] Tim Bass. Mythbusters: Event stream processing versus complex event processing. In *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 1–1, New York, NY, USA, 2007. ACM.

[3] Norman W. Paton and Oscar Díaz. Active database systems. In *ACM Comput. Surv.* ACM, 1989.

[4] Dennis McCarthy and Umeshwar Dayal. The architecture of an active database management system. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 215–224, New York, NY, USA, 1989. ACM.

[5] Narain H. Gehani, H. V. Jagadish, and Oded Shmueli. Composite event specification in active databases: Model & implementation. In *VLDB '92: Proceedings of the 18th International Conference on Very Large Data Bases*, pages 327–338, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[6] Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S. K. Kim. Composite events for active databases: Semantics, contexts and detection. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile proceedings*, pages 606–617, Los Altos, CA 94022, USA, 1994. Morgan Kaufmann Publishers.

[7] Antony Galton and Juan Carlos Augusto. Two approaches to event definition. In *DEXA '02: Proceedings of the 13th International Conference on Database and Expert Systems Applications*, pages 547–556, London, UK, 2002. Springer-Verlag.

[8] Raman Adaikkalavan and Sharma Chakravarthy. Snoopib: Interval-based event specification and detection for active databases. *Data Knowl. Eng.*, 59(1):139–165, 2006.

[9] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. *SIGMOD Rec.*, 21(2):81–90, 1992.

[9] Consistency of a rule base assumes that a rule base does not contain any kind of anomalies like inconsistent, redundant or circular rules.

[10] Stella Gatziu and Klaus R. Dittrich. Detecting composite events in active database systems using petrinets. In *Proc. Fourth International Workshop on Active Database Systems Research Issues in Data Engineering*, pages 2–9, 1994.

[11] Sharma Chakravarthy. Sentinel: An object-oriented dbms with event-based rules. In Joan Peckham, editor, *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 572–575. ACM Press, 1997.

[12] Michael A. Covington, Donald Nute, and André Vellino. *Prolog programming in depth*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1987.

[13] Franois Bry and Michael Eckert. Towards formal foundations of event queries and rules. In *Second Int. Workshop on Event-Driven Architecture, Processing and Systems EDA-PS*, 2007.

[14] A. Paschke, A. Kozlenkov, and H. Boley. A homogenous reaction rules language for complex event processing. In *International Workshop on Event Drive Architecture for Complex Event Process*, 2007.

[15] U. Dayal, A. P. Buchmann, and D. R. McCarthy. Rules are objects too: A knowledge model for an active, object-oriented databasesystem. In *Lecture notes in computer science on Advances in object-oriented database systems*, pages 129–143, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[16] H. Behrends. An operational semantics for the activity description language adl. Technical report, Universität Oldenburg, June 1994. Technical Report TR-IS-AIS-94-04.

[17] Charles L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[18] D.P. Miranker. *TREAT: a new and efficient match algorithm for AI production systems*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1990.

[19] Don Batory. The leaps algorithms. Technical report, Austin, TX, USA, 1994.

[20] Sharma Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data Knowl. Eng.*, 14(1):1–26, 1994.

[21] Erik Behrends, Oliver Fritzen, Wolfgang May, and Franz Schenk. Combining eca rules with process algebras for the semantic web. In *RuleML*, 2006.

[22] Milner R., editor. *Calculus of Communicating Systems*. Theoretical Computer Science, 1983.

[23] Wing J.M. Faq on pi-calculus. In *Microsoft Internal Memo*, 2002.

[24] Bruno Berstel. Extending the rete algorithm for event management. In *Proc. Ninth International Symposium on Temporal Representation and Reasoning TIME 2002*, pages 49–51, Washington, DC, USA, 7–9 July 2002. IEEE Computer Society.

[25] Michael Kifer, Arthur Bernstein, and Philip Lewis. *Database Systems - An Application-Oriented Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2006.

[26] Asaf Adi, Ayelet Biger, David Botzer, Opher Etzion, and Ziva Sommer. Context awareness in amit. In *Active Middleware Services*, 2003.

[27] Anicic D. Stojanovic N. Towards creation of logical framework for event-driven information systems. In *To appear in: 10th International Conference on Enterprise Information Systems*, 2008.