Evolving Triggers for Dynamic Environments

Goce Trajcevski[1*] Peter Scheuermann[1**] Oliviu Ghica[1] Annika Hinze[2] Agnes Voisard[3]

[1] Northwestern Univ., Dept. of EECS, {goce,peters,oliviu}@eecs.northwestern.edu
[2] Univ. of Waikato, Computer Science Dept., hinze@cs.waikato.ac.nz
[3] Fraunhofer ISST and FU, Berlin, Agnes.Voisard@isst.fhg.de

Abstract. In this work we address the problem of managing the reactive behavior in distributed environments in which data continuously changes over time, where the users may need to explicitly express how the triggers should be (self) modified. To enable this we propose the $(ECA)^2$ – Evolving and Context-Aware Event-Condition-Action paradigm for specifying triggers that capture the desired reactive behavior in databases which manage distributed and continuously changing data. Since both the monitored event and the condition part of the trigger may be continuous in nature, we introduce the concept of *metatriggers* to coordinate the detection of events and the evaluation of conditions.

1 Introduction and Motivation

Many application domains deal with data that changes very frequently and is generated by distributed and heterogeneous sources. These data-properties have spurred extensive research efforts in several fields. In Event-Notification Systems (ENS), and Publish-Subscribe (P-S) systems [3, 11], typically an instance user's profile is matched against the current status of continuously evolving data sources, and appropriate notifications are sent to the user. The main focus of Continuous Queries (CQ) processing [6, 12] is on efficient management of user queries over time, without forcing the users to re-issue their queries. The data values may arrive as streams which the system has to process on the fly [5, 4, 14] and, furthermore, the data may be multidimensional in nature, as is the case in Location-Based Services (LBS) [16] and Moving Objects Databases (MOD) [10]. In some applications, e.g sensor networks, the data management must consider other constraints such as the limited battery-lifetime of the nodes [22].

One may observe that in the majority of the applications, there is a need for some form of a *reactive* behavior. The database community has provided many results on the topic of Active Databases (ADb), which manage triggers operating under the Event-Condition-Action (ECA) paradigm [8, 15, 20]. In the recent years there have been works incorporating ECA-like triggers in novel, highly-heterogeneous, distributed and dynamic data-driven application domains, e.g.,

* Research supported by the Northrop Grumman Corp., contract: P.O.8200082518
** Research supported by the NSF grant, contract: IIS-0325144/003

the Web [9], peer-to-peer (P2P) systems and sensor networks [22]. Despite the co-existence of the large body of works in ENS, CQ, Data Streams, MOD [3, 5, 4, 6, 12, 11, 16, 10], all of which have the common need of dealing with dynamically changing information, and the rich history of ADb results [8, 15, 20] – there is a lack of tools that would enable using the "best of all the worlds". Namely, there is no paradigm that allows the users to seamlessly tie: (1) Detection of (composite) events obtained by monitoring continuously changing data with (2) Evaluation of conditions that are continuous queries and with (3) Dynamical adjustment of the triggers themselves – all for the purpose of executing a desired policy in a constantly evolving domain of interest.

In order to illustrate our motivation better, we present two scenarios and we analyze the requirements posed by each of them.

I. Rq1: "When *a moving object is continuously_moving_towards the region R for more than 5 minutes,* if *there are less than 10 fighter jets in the base B1,* then *send* `alert_b` *to the armored unit A1.* *Also* *send* `alert_a` *to the infantry regiment I1,* when that *object is closer than 3 miles to R,* if *all the marine corps units are further then 5 miles from R".*

Rq1 needs to detect a composite event *(moving continuously towards...)*, using the individual *(location,time)* updates as simple events. These can be obtained, e.g., by tracking sensors [22], and in [18] we provided efficient algorithms for detecting the *continuously moving towards* predicate. **RQ1** also needs to initiate a continuous query at a remote system – the one monitoring the status of the air-base $B1$. However, **Rq1** has some other subtleties:

• It needs the status of the air-base $B1$ for as long as the original enabling event *moving towards* is still valid. After detecting its enabling event, **Rq1** requires that the system "spans" its attention to monitoring one more event *(closer than 3 miles to R)* and, upon its detection, request an evaluation of another remote condition-query, which happens to be instantaneous. Observe that there is a *binding* between the new event to the original event – the new one needs to focus on the distance pertaining to the particular object that satisfied the original enabling event.

II. Rq2: "When *the IBM stock in New York stock exchange has three consecutive increases of its value within 30 minutes with a total increase of at least 5%,* if *there is a stock exchange at which both IBM and Intel stocks within 15 min. from the originating event have achieved a one hour interval without dropping, then* execute portfolio P1 *for purchasing IBM shares at* that *stock exchange.* *Otherwise,* if *there is a non-decrease of the Google stock for 45 minutes, while the IBM increase is still valid on* any other *stock exchange,* execute portfolio P2 *for purchasing IBM shares at* that other *stock exchange.* *Subsequently, only* execute portfolio B *for purchasing Motorola shares,* when *its stock has two consecutive increases by a total of at least 8% in London,* if *its average daily increase on any other exchange market is non-negative".*

• Unlike **Rq1**, now the validity of the composite event related to the IBM values, which is detected based on the primitive events that are updates of the value of the its stock, is limited by an explicit time-value – *15 min.* The system requests

an evaluation at another site of the if condition, however, this condition is peculiar in that it combines a continuous query with the one-hour past history of the system [5] of the IBM and Intel stocks, but allows for a portion of that history to be satisfied within the continuous query itself, for as long as it is within 15 minutes after the detection of the IBM-increase event.

• In **Rq2** the user has an "alternative plan" of reacting, if the first condition fails. This alternative depends on the outcome of another continuous query *(Google stock)*, which is tying the duration of interest for evaluating the continuous query with the "native" enabling event. **Rq1** requires the system to span its attention, but **Rq2** in its last part requires the system to completely *shift* the focus of its reactive behavior. After the failure of the respective Intel and Google-related criteria, the user is no longer interested in reacting to the events related to the increases of the IBM stock and wants to focus on the Portfolio B for Motorola shares.

The observations related to **Rq1** and **Rq2** have motivated our research towards the new paradigm for reactive behavior. Our main contributions are:

• We introduce a paradigm for specifying reactive behavior, called $(ECA)^2$ (Evolving and Context-Aware Event-Condition-Action), that enables the users to specify triggers that pro-actively evolve so that they can ensure a desired policy.

• We introduce the concept of a *metatrigger* for the purpose of minimizing the communication overhead and ensuring behavioral correctness in distributed setting. We observe that there is a duality in the nature of events and conditions that can be exploited in the functioning of the the metatriggers.

In the rest of this paper, Section 2 introduces the $(ECA)^2$ paradigm and its syntactic elements. The concept of the metatriggers is presented in Section 3, and Section 4 concludes the paper.

2 Evolution of the Triggers

In this section, we explain the main aspects of the specification of the triggers under the $(ECA)^2$ paradigm. The syntactic components are presented in Figure 1. Firstly, observe that in the events, conditions and actions we allow variables to be used. Thus, for example, $E_p(EV_p)$ denotes that the event of the parent-trigger E_p has the (vector of) variable(s) EV_p in its specification; similarly, $C_{p1}(VC_p1)$ denotes the query and the variables used in the first condition of the parent trigger. We assume that the usual rules for *safety* [19] of the variables apply, in the sense that each variable that appears in a negative literal, must also appear in a positive literal, or have a ground value at the time of the invocation/evaluation of the corresponding (negated) predicate. Secondly, observe that we allow two types of children-triggers to be specified within the scope of a given (parent) trigger. As is commonly done in the programming languages, we use rectangles to visualize the nesting of the relative scope of children-triggers within the scope of the parent-trigger. As indicated in Figure 1, the user can specify an arbitrary level of nesting of descendants within the children-triggers.

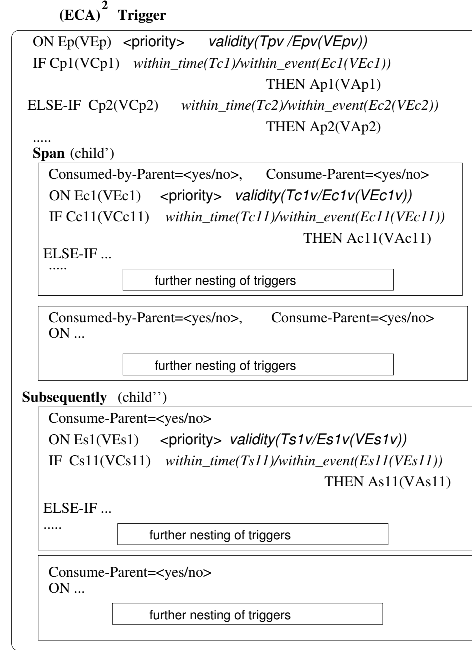

Fig. 1. Evolving Triggers Specification

Before we give detailed explanation of the syntax of the $(ECA)^2$ triggers, we provide an example for the **Rq2**, illustrated in Figure 2. Using variables, one can express the desired relationships among the locations of the stock exchanges for evaluating the criteria of interest. Thus, when evaluating the condition for the alternative policy regarding the *Google* stock, the variable SE_2 denotes a stock exchange which is different from *New York*. By using $SE2$ as a variable in the action that executes the Portfolio $P2$, we ensure that the purchase is executed at "that other" stock exchange. The important observation here is that whenever the value of the IBM stock in New York stock exchange decreases it terminates the validity of the enabling event for the (parent) trigger. Past that point, the *child* trigger which implements the reactive policy for Motorola stock exists on its own, monitoring its respective event. Now we proceed with explaining the elements in the syntax of the $(ECA)^2$ triggers:

1. The option `validity` in the trigger's specification allows the user to state *how long* should the trigger be considered "alive". It reflects the user's policy, and it can be either an explicit time-value, or an event which, when detected causes the particular trigger's instance to be disabled. As a special case, one is able to specify *for as long as the original enabling event is valid*, by utilizing proper expressions of the available event algebra. For example, in the case of **Rq2** one may specify a composite event which is (a sequence of) IBM-increase events that enabled the trigger, followed by an IBM-decrease related event.

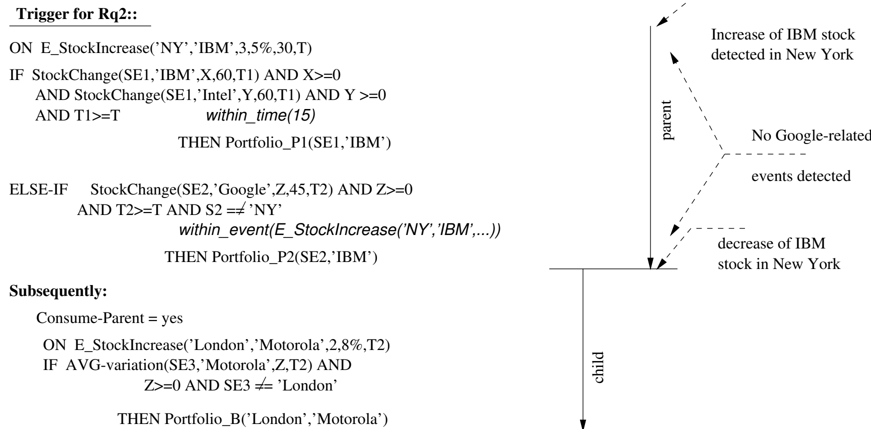

Fig. 2. Example Trigger for **Rq2**

2. The *Else-If* parts at each level of nesting of the triggers correspond to alternative policies, based on the value of the respective conditions, once an instance of a particular trigger is "awaken". Clearly, these can be written as conditions of separate triggers with the same enabling event. We use the way depicted in Figure 1 for compactness, assuming that the ordering of the conditions actually corresponds to the users' preference when it comes to their evaluation. Some prototype systems (e.g., Starburst [21]) enable the users to explicitly state their ranking for (partial) ordering among the triggers (e.g., `PRECEDES`), but some commercial systems conforming to the ANSI standard (e.g., Oracle 9i [2]) do not allow this – triggers are ranked based on their time-stamps and that ranking is not always ensured at run-time. As indicated in Figure 1, we do consider the option of an explicit numeric priority specification for the triggers, which can be straightforwardly extended to the conditions.

3. Each condition has two options for indicating for how long its corresponding continuous query should be evaluated. One option is to explicitly list a time-interval value, as commonly done in CQ systems (e.g., [6]). An example for this is the statement *within_time(15)* in the trigger for **Rq2**. The other option is to specify an event which will confirm the termination of the user's interest in that condition. In the case of **Rq1**, the user is interested in getting updates about the state of the air-base for as long as the composite event *E_moving_towards* is satisfied, based on the *(location,time)* update-events [18].

4. There are two types of *child-triggers*:

4.1. The first type – *child'*, enables a reaction to subsequent occurrences of other events that could potentially request monitoring of other conditions. This is the case in **Rq1**, where the user is also interested in detecting the proximity of that particular object to the region R. The value *Consumed-by-Parent = yes* indicates that the child-trigger should terminate when the parent-trigger terminates. Conversely, *Consumed-by-Parent = no*, specifies that the child-trigger

should continue its execution even though the parent has ceased to exist. As an example, in **Rq1** the value *Consumed-by-Parent = no* specifies that, although the particular moving object may no longer be *moving_continuously_towards* the region R (e.g., it is following a zig-zag route), which disables the original (parent) trigger, the user is still interested in monitoring the *distance* of that particular object. Both consumption parameters provide means to dynamically `enable` and `dissable` instances of the triggers.
4.2. The second type of a child-trigger – *child"*, is specified with the **Subsequently** option, and its intended meaning is that, after the particular parent-trigger has been enabled, and all its "options have been exhausted" (e.g., expiration of the interval of interest for the continuous queries; no occurrence of the events for the *child'*-triggers), the user wants to focus on other aspects of the possible subsequent evolutions of the domain of interest. In the case of **Rq2**, the user shifts his interests to the properties related to the *Motorola* stock. However, the user has the option of stating whether in the future, the system should consider "waking-up" the parent trigger again or not. This is achieved by the statement *Consume-Parent*. *Consume-Parent = yes* reflects the user's intention not to consider the parent-trigger in the future at all. In the context of our **Rq2** example, the user does not want to bother with the future variations related to the IBM. In a sense, this is an equivalent to the SQL `drop` trigger rule, as no further instances of the parent-trigger are desired. *Consume-Parent = no* has the opposite effect.
Having the instances of the child-triggers active is similar in effect to the SQL `enable` command. However, in practice it is very unlikely to expect that the desired behavior can be achieved if the users are to manually execute it. Furthermore, attempting to write a child-trigger as a separate trigger from (and at the same scoping/nesting level as) its parent, with an enabling event which is a sequence of the *parent_event* followed by the *own_event*, may yield an unintended behavior. For example, in **Rq2** if, instead of being a child, the trigger related to *Motorola* stock is specified independently, with the event *E_StockIncrease('NY', 'IBM',3,...)* **;** *E_StockIncrease('London','Motorola',2,...)*, the user may end up executing the Portfolio B in the settings in which, according to his preferences, he should have executed portfolio P1. The reason for this is that the condition of the *Motorola*-related trigger is an *instantaneous* query, which may be satisfied as soon as the composite event which enables the corresponding trigger is detected.

3 Metatriggers

The *metatrigger* is a module that is in charge of coordinating the detection of events and evaluation of the conditions in distributed environments, in a manner that ensures behavioral correctness and minimizes the communication overhead. To better motivate it, observe the following detailed example in the context of **Rq1** assuming, for the sake of argument that the *(location,time)* are detected every two minutes.

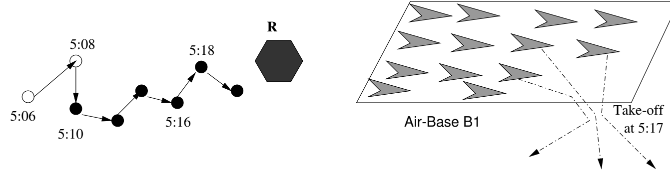

Fig. 3. Dynamics of Events and Conditions

As illustrated in Figure 3, the system began monitoring the object at 5:06, however, the *(location,time)* updates at 5:06 and 5:08, depicted with blank circles, were discarded because they were of no use for detecting the event of interest. Starting at 5:10, the system can detect the occurrence of the desired composite event *moving towards*) at 5:16 [18] which, in turn, "awakes" an instance of the corresponding trigger for **Rq1**. Upon checking the condition (*less than 10 airplanes in B1*), the system will find out that there are actually 12 airplanes there and will not execute the action part *(alert)*. However, as illustrated in Figure 3, in a very short time-span, three jets have left the air-base and, by 5:17 it has only 9 jets available. Intuitively, the trigger for the **Rq1** should fire and raise the alert. However, this may not happen until 5:18 at which time the event *moving towards* is (re)detected. In many time-critical applications, this may cause unwanted effects. One possible solution is to periodically poll the remote database, however, this may incur a lot of unnecessary communication overhead[4] and, moreover, may still "miss" the actual time-point at which the condition became satisfied.

The main role of the metatriggers is the management of the type of behavior as described above in distributed environments. Figure 4 illustrates the position of the metatrigger module in the context of a typical ADb architecture, extended with an Event-Base (EB) (c.f. [8]). The arrowed lines indicated the data flow among the modules. Note that the module for the Continuous Queries Processing (CQP), is coupled with the Query Processing (QP) and the Rule (trigger) Processing (RP) modules [6].

When it comes to managing the reactive behavior in distributed settings, the crux of the metatriggers is the *Event and Conditions Manager* (ECM) component. This component translates the original specifications of the user and generates a new set of triggers, events and conditions that achieve the desired behavior, but are much more efficient for distributed environments. To describe this task more formally, consider the following simplified version of a trigger:

TR1: ON E_1
IF $C_{1i} \wedge C_{1c}$
THEN A_1

Its condition part consists of two conjuncts:

• C_{i1}-instantaneous conditions, whose evaluation may be bound to various states.

[4] Observe that the user may insist on a particular frequency of re-evaluation of the continuous query (c.f. [6]).

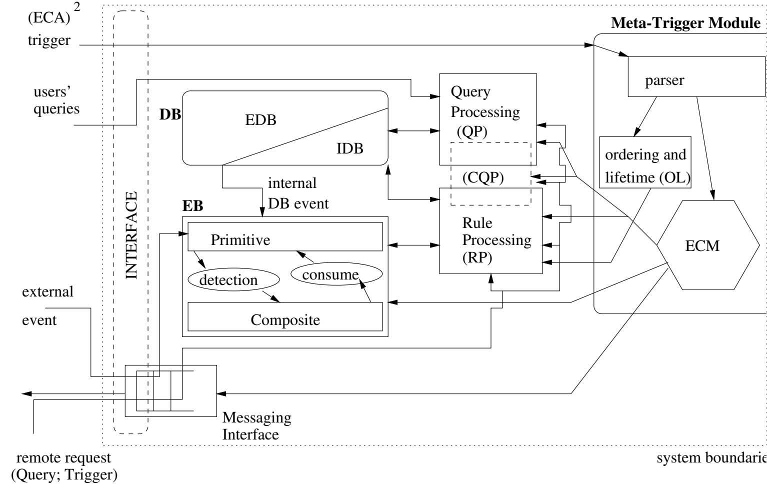

Fig. 4. Metatriggers Activities

Such bindings have already been identified as a *semantic dimension* of the active databases [8, 15] and there are syntactic constructs that can specify particular states for evaluating the condition of the triggers (e.g., *referencing old/new*).
• C_{1c}- a condition which expresses a continuous query.
The ECM component of the metatrigger performs the following activities:
1. Translate the specifications of the original trigger into:

$$\begin{array}{ll} \textbf{TR1':} & \textbf{ON} \quad E_1 \ \textbf{;} \quad (E_{\overrightarrow{C_{1c}}} \ \textbf{;} \quad E_{\overleftarrow{C_{1c}}}) \\ & \textbf{IF} \quad C_{1i} \\ & \textbf{THEN} \quad A_1 \end{array}$$

Where $E_{\overrightarrow{C_{1c}}}$ and $E_{\overleftarrow{C_{1c}}}$ are two new events with the following semantics:
1.1. $E_{\overrightarrow{C_{1c}}}$ – an event denotes the request for evaluating C_{1c}, which may have to be sent to a remote site – e.g., in the case of **Rq1** it is send to the air-base.
1.2. $E_{\overleftarrow{C_{1c}}}$ – an event (external in case of **Rq1**), which denotes that the continuous condition has been evaluated to true, and the notification about it has been received. Observe that the new local trigger **TR1'** is now enabled by the composite event which is the sequence of events E_1 **;** $(E_{\overrightarrow{C_{1c}}}$ **;** $E_{\overleftarrow{C_{1c}}})$
2. It translates the continuous query C_{1c} of the condition into:
2.1. A message for the remote site, requesting immediate evaluation and notification if true;
2.2. A trigger that is transmitted to the remote site, which essentially states:

$$\begin{array}{ll} \textbf{TR1c:} & \textbf{ON} \quad E_1; \ E_{\overrightarrow{C_{1c}}} \\ & \textbf{IF} \quad C_{1c} \\ & \textbf{THEN} \quad \textbf{A}(\text{Send_Notification}(E_{\overleftarrow{C_{1c}}})) \end{array}$$

3. Lastly, the ECM generates the specification of another local trigger **TR1"**, whose description we omit – but whose purpose is to detect when the original trigger **TR1**, as specified by the user, has "expired", i.e., the criteria used in the

`validity` specification, temporal or event-based, is satisfied, and:
3.1. `disable` the current instance of the local trigger **TR1'**.
3.2. Send a notification that the instance of the **Tr1c** in the remote site should be `disabled`.

What we described above exemplifies how something that was initially perceived as a pure query-like condition, becomes a "generator" of a several new events/triggers. We only explained the basic functionality of the ECM as a translator for a simplified version of the original specifications of the user's trigger. Clearly, in reality, one may expect more sophisticated queries whose translation and generation of the equivalent new events, triggers, and messages to the remote sites will be more complicated. In the settings of the **Rq1**, one may observe another motivation for translating the original condition's query: the predicate `JetsCount` (c.f. Figure 2), say, for security purposes, may be a *view* and the user cannot express much at the specification time of the corresponding trigger.

Although the ECM is the most relevant component of the metatrigger module, it has few other components. The parser extracts the constructs of the syntax that define the corresponding events and conditions, as well as user's preferences for priority/ordering. The *Ordering and Lifetime* (OL) component of the metatrigger works in conjunction with the RP component. It ensures that, whenever a particular event is detected, the order of evaluating the conditions and executing the actions among all the triggers "awaken" by that event, conforms with the user's specifications. We re-iterate that although some prototype ADb systems, such as Starburst [20] provide the option for priorities among the triggers ([8, 15]), the commercial DBMS with active capabilities, conforming with the ANSI SQL99 standard specifications [1] do not. Since we are using Oracle 9i [2], we needed to write a separate PL/SQL routine (c.f. [17]). The OL component of the metatrigger is also in charge of `enable`-ing the (instances of the) child-triggers in the proper states of the evolution of the system. Upon "ceasing" of a particular trigger, OL ensures that the appropriate clean-up actions are performed which, based on the values of the *Consumed-by-Parent* and *Consume-Parent* parameters, are either `disable` or `drop`.

4 Related Works and Concluding Remarks

There is a large body of existing results in several research areas that address managing of (re)active behavior [3, 5, 7, 8, 11, 13, 15, 21, 22]. These works provide a technical foundations for, and in turn, can benefit from our work, however, their detailed discussion is well beyond the scope of this paper.

We presented a novel paradigm $(ECA)^2$, for triggers that are aware of the dynamic correlation between the events and conditions and, in a sense, can "react in a proactive manner" – by modifying themselves. We also provided syntactic constructs for specifying the triggers under this paradigm and enable using the $(ECA)^2$ paradigm in dynamic distributed environments, and we proposed the concept of the metatriggers as a possible tool for their management. Currently, we are focusing on further incorporating the $(ECA)^2$ in heteroge-

neous/multidatabase settings, and we would like to believe that, in a near future, our work will motivate a wide spectrum of new challenges, ranging from theoretical aspects (e.g., termination/expresiveness [21]) up to intricate details that depend on the application/problem domain constraints (e.g., power-limitations in sensor networks [22]; interplay among context variables in LBS [16]).

References

1. ANSI/ISO International Standard: Database language SQL. http://webstore.ansi.org.
2. Oracle 9i. www.oracle.com/technology/products/oracle9i.
3. A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM-TOCS*, 19(3), 2001.
4. S. Chandrasekaran and M.J. Franklin. Streaming queries over streaming data. In *VLDB Conference*, 2002.
5. S. Chandrasekaran and M.J. Franklin. Remembrance of streams past: Overload-sensitive management of archived streams. In *VLDB Conference*, 2004.
6. J.J. Chen, D.J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *ACM SIGMOD Conference*, 2000.
7. Y. Diao, S. Rizvi, and M.J. Franklin. Towards an internet-scale XML dissemination service. In *VLDB Conference*, 2004.
8. P. Fraternali and L. Tanca. A structured approach for the definition of the semantics of active databases. *ACM TODS*, 20(4), 1995.
9. G.Papamarkos, A.Poulovassilis, and P.T.Wood. Event-condition-action rule languages for the semantic web. In *SWDB Workshop (at VLDB)*, 2003.
10. R.H. Güting and M. Schneider. *Moving Objects Databases*. Morgan Kaufmann, 2005.
11. A. Hinze and A. Voisard. Location-and time-based information delivery in tourism. In *SSTD*, 2003.
12. L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE-TKDE*, 11(4), 1999.
13. S. Madden, M.A. Shah, J.M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *ACM SIGMOD Conference*, 2002.
14. C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *ACM SIGMOD*, 2003.
15. N. Paton. *Active Rules in Database Systems*. Springer-Verlag, 1999.
16. J. Schiller and A. Voisard. *Location-based Services*. Morgan Kaufmann Publishers, 2004.
17. G. Trajcevski, H. Ding, and P. Scheuermann. Context-aware optimization of continuous range queries for trajectories. In *MobiDE Workshop (at SIGMOD)*, 2005.
18. G. Trajcevski, P. Scheuermann, H. Brönnimann, and A. Voisard. Dynamic topological predicates and notifications in moving objects databases. In *MDM*, 2005.
19. J. D. Ullman. *Principles of Database and Knowledge – Base Systems*. Computer Science Press, 1989.
20. J. Widom. The Starburst active database rule system. *IEEE TKDE*, 8(4), 1996.
21. J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.
22. F. Zhao and L. Guibas. *Wireless Sensor Networks: an Information Processing Approach*. Morgan Kauffman, 2004.