The Convergence of AOP and Active Databases: Towards Reactive Middleware

Mariano Cilia, Michael Haupt, Mira Mezini, and Alejandro Buchmann

Department of Computer Science
Darmstadt University of Technology - Darmstadt, Germany
{cilia,haupt,mezini,buchmann}@informatik.tu-darmstadt.de

Abstract. Reactive behavior is rapidly becoming a key feature of modern software systems in such diverse areas as ubiquitous computing, autonomic systems, and event-based supply chain management. In this paper we analyze the convergence of techniques from aspect oriented programming, active databases and asynchronous notification systems to form reactive middleware. We identify the common core of abstractions and explain both commonalities and differences to start a dialogue across community boundaries. We present existing options for implementation of reactive software and analyze their run-time semantics. We do not advocate a particular approach but concentrate on identifying how the various disciplines can benefit from each other. We believe that AOP can solve the impedance mismatch found in reactive systems that are implemented through inherently static languages, while AOP can benefit from the active database community's experience with event detection/composition and fault tolerance in large scale systems. The result could be a solid foundation for the realization of reactive middleware services.

1 Introduction

Software development in the past decade has exhibited several trends:

– monolithic ad-hoc software development is being replaced by service-based architectures relying on customizable generic services;
– crosscutting concerns must be modularized and are often added after a system has been designed;
– the need for asynchronous interactions has been recognized, particularly in the face of mobility and the resulting instability of communications;
– the need for reactive behavior and the ability to handle exceptions are essential for very diverse applications ranging from ubiquitous computing to event-based supply chain management.

Solutions to cope with (some of) these trends have emerged in different communities: the database community has developed active databases, the programming languages community has developed aspect oriented programming, and the middleware community has developed asynchronous notification mechanisms and service-based architectures. Each approach reflects the idiosyncracies of the corresponding community, but they all share a core of common abstractions extended by issues that are important in a given community as shown in Fig. 1.

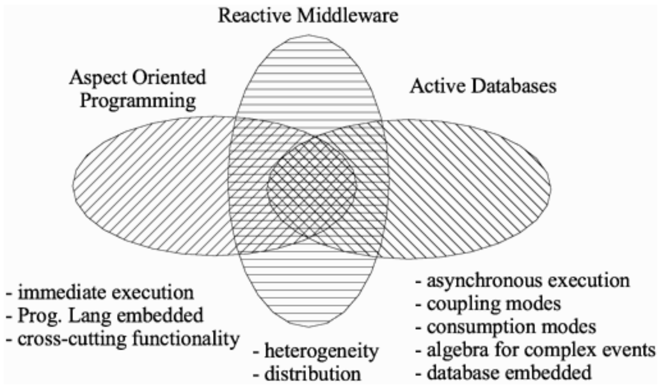

Fig. 1. Technologies in context

It is the goal of this paper to identify the common abstractions, understand both similarities and differences, and start a dialogue across community boundaries. Towards this goal, the first contribution of this paper is to introduce the terminology, to identify the commonalities of the various approaches to reactive software, present experiences and discuss the difficulties. We exploit the fact that each of the authors is thoroughly familiar with at least two of the three areas and has good working knowledge of the third.

The second goal of this paper is to present existing options for implementation of reactive software and to analyze their run-time semantics. The flexibility of reactive systems is determined by the time at which reactive functionality can be added to existing code, how the reactive capability can be activated or deactivated, and how an event, i. e., the occurrence of a happening of interest, is signalled. To this end we introduce a spectrum ranging from compile-time to run-time binding on one dimension, and from non-invasive to invasive situation detection mechanisms. Invasive detection mechanisms require the modification of the underlying code, while non-invasive mechanisms do not.

In this paper we do not push any particular approach but concentrate on identifying how the various disciplines can benefit from each other. We believe that aspect oriented programming can solve the impedance mismatch found in active databases and service based middleware when dynamic capabilities are implemented through inherently static languages. On the other hand, aspect oriented programming can benefit from the experience gained in the active database area with event detection/composition and fault tolerance in large-scale systems.

The result could be a solid foundation for the realization of reactive middleware services and eventually a platform for reactive software.

The remainder of this paper presents in Section 2.1 the underlying technologies and introduces the terminology. Section 2.2 deals with the definition of reactions. Section 2.3 deals with the execution of reactions, i. e., the run-time semantics. Section 2.4 discusses the options afforded by the various run-time environments. Section 3 shows how the technologies converge towards reactive middleware. Finally, Section 4 presents conclusions and a brief summary of our ongoing work.

2 Active Databases and AOP Side-by-Side

2.1 Origins and Basic Concepts

The basic problem that motivated the work in active databases and AOP is modularization of crosscutting concerns and the need to detect relevant situations and react to them efficiently. In the case of monitoring applications that track changes to the underlying (passive) database the problem lies in the need to poll periodically the state of the database to detect changes. This is highly inefficient, not only because many unnecessary queries are executed but also because queries are issued in user space, the queries in their execution traverse several layers of the database management system (DBMS) and monitoring functions (conditions) may be replicated across applications. Active databases solved this problem by defining *Event-Condition-Action* (ECA) rules that are defined in the schema and executed in the DBMS kernel. Whenever an event is detected and a guarding condition is true, the corresponding action is executed. *Aspect Oriented Programming* (AOP) has been motivated by the need to modularize crosscutting behavior, such that new crosscutting behavior can be added in a localized manner and without changing existing code. AOP introduced join points and point cuts to signal situations of interest and advice to react to them. In the remainder of this section we summarize the notions of Active Databases and Aspect Oriented Programming, and introduce the terminology used throughout this paper.

Active Databases: In conventional (*passive*) database management systems data is created, modified, deleted and retrieved in response to requests issued by users or applications. In the mid-80s database technology was confronted with monitoring applications that required the reaction to changes in the database. To avoid wasteful polling, passive database functionality was extended with ECA rules to allow the database system itself to perform certain operations automatically in response to the occurrence of predefined events. Those systems are known as *active DBMSs* [1, 63, 53]. They are significantly more powerful than passive DBMSs since they can (efficiently) perform functions that in passive database systems must be encoded in applications. The same mechanism can also be used to perform tasks that require special-purpose subsystems in passive databases

(e. g., integrity constraint enforcement, access control, view management, and statistics gathering).

Historically, production rules were the first mechanism used to provide automatic reaction functionality. Production rules are Condition-Action rules that do not break out the triggering event explicitly. Instead, they implement a polling-style evaluation of all rule conditions. In contrast, ECA rules explicitly define triggering events, e. g., the fact that an update or an insert occurred, and conditions on the content of the database are only evaluated if the triggering event is signaled.

ECA rules consist of three parts: a lightweight *Event* causes the rule to be fired; an (optional) *Condition* is checked when the rule is fired; and an *Action* is executed when the rule is fired and its guarding condition evaluates to true.

In active relational databases, events were modelled as changes of state of the database, i. e., insert, delete and update operations that could trigger a reaction [31, 59]. This basic functionality is common fare in today's commercial DBMSs in the form of triggers. In object-oriented systems, ECA rules are considered as *first-class objects* [17]. This treatment of ECA rules as first-class entities enables them to be handled homogeneously like any other object. In these systems more general events were defined: temporal, method invocation (with before and after modifiers), and user-defined events [17, 3, 28, 11, 66].

In addition to these events, more complex situations that correlate, aggregate or combine events can be defined. This is done by using an event algebra [28, 67] that allows the definition of *composite or complex events*.

Active database functionality developed for a particular DBMS became part of a large monolithic piece of software (the DBMS). As it is well known, monolithic software is difficult to extend and adapt. Moreover, tightly coupling the active functionality to a concrete database system precludes its adaptation to today's Internet applications, where heterogeneity and distribution play a significant role but are not directly supported by (active) database systems.

Aspect Oriented Programming: The goal of aspect-oriented programming is to facilitate the separation and modularization of concerns whose natural structures are crosscutting [46, 47]. Crosscutting concerns imply different decompositions of the system into modules. The different decompositions yield different models, i. e., sets of interrelated modules. Given two crosscutting models M1 and M2, if we adopt one of them, say M1, as our basic decomposition of the system, then definitions that are modularly captured in M2 will be spread around several modules in M1, causing code scattering and tangling, negatively affecting the extensibility and maintainability of the software. For a technical and an intuitive definition of the term "crosscutting models", the reader is referred to [46], respectively [47].

To avoid such scattering and tangling, with AOP the question is not which criteria to choose for decomposing the system, but how to support the decomposition according to several independent criteria simultaneously. Aspect-oriented languages modularize a system's various crosscutting models into independent aspect modules and provide appropriate means for specifying *when* and *how* they should join the overall definition of the system. By modularizing crosscutting concerns into aspects, behavior that conventional programming would distribute throughout the code congeals into a single textual structure, making both aspect code and the target easier to understand [26]. This promises simpler system evolution, more comprehensible systems, adaptability, customizability, and easier reuse.

The design space of aspect-oriented languages is determined by several factors. First, AO languages may vary in how they represent the modules they support for specifying individual crosscutting models. Second, different approaches can be taken for the specification of *when* crosscutting models join. This specification is expressed using *join points* that can be placed in the source code of different aspects, or they can be points in the object call graph of the running system, yielding the distinction between AO languages with a static versus dynamic join point model. Furthermore, one might distinguish between coarse-grained join points such as method signatures, and fine-grained join points like the access of a variable within the execution of a control flow. Finally, AO languages may also vary in the specification of *how* aspects come together at join points.

Implementation approaches for AO languages can be classified based on two criteria. First, systems may use different ways for detecting/signaling join points. The second criterion regards how actual implementations perform the dispatch to advice code at join points once they are detected.

In the course of this paper, we will refer to the AspectJ system [41, 5] because it offers a rather well-defined terminology for the most important parts of aspect orientation and is the most prominent AO language today.

2.2 Definition of Situations and Reactions

Both AOP and active databases provide mechanisms for describing situations under which specific reactions should be invoked. Although the concepts and mechanisms are surprisingly similar, the two communities use different terminology. In this section we describe the basic detection and reaction primitives from the point of view of their specification and show the convergence of both technologies.

Defining ECA Rules: Primitive or composite events are associated with ECA rules in order to describe the situation that must be detected for the action to be executed. The subsystems of an active database - query processor, transaction and recovery managers, etc. – can signal the occurrence of primitive events [9]. These can be data manipulation operations, timer events or explicit events signaled from outside the database, e. g., from the operating system.

The use of method invocation events in the definition of ECA rules can include `before`, `after`, and `instead` modifiers. These are used to explicitly define whether the rule should be executed before or after a method execution or alternatively to the original code.

An event is characterized by having a type and a set of parameters. Parameters are associated to event occurrences with the purpose of describing information about the context where the event was signaled. These parameters are specific to the event type. For instance, an occurrence of a database event of type `insert` carries the following parameters: name of table in question, the involved fields and their corresponding values, transaction identification, timestamp, etc. Both a rule's condition and its action have access to the associated parameter information of an event occurrence.

As mentioned before, events may be composed to form more complex events by using an event algebra. Once a pattern of events being signaled matches an event expression, the associated rule is fired. A variety of event algebras has been defined [17, 30, 28, 13, 67]. Examples of possible event expressions are:

- **Sequence:** The two events must occur in the given order for the expression to match.
- **Alternative:** One of the two events must occur for the expression to match.
- **Closure:** The event may occur multiple times during a transaction, but it is not effectively signaled until the transaction ends or until it has occurred a certain number of times. The event's context information is accumulated.

An important issue must be considered when composing events, specifically when more than one event occurrence is to be consumed in the composition of a complex event. The specification of a *consumption mode* [13] associated to the event part of the rule specification allows the rule processing engine to precisely determine which event occurrences must be considered for consumption. Different consumption modes may be required by different application domains. The two most common modes are `recent` and `chronicle`. In the former, common in sensor-driven applications, the most recent event occurrences of the specified type are used, while in the latter, common in workflow applications, the oldest event occurrences of the specified type are consumed.

Conditions are optional and allow the expression of additional constraints that must be met for the action to execute. If they are missing we speak about *Event-Action (EA) rules*. Conditions can be expressed as Boolean predicates or query language expressions. Also external method invocations can be used. The specification of conditions can involve variables that will be bound at run-time with the content of triggering events.

The action part of an ECA rule may be a sequence of operations related to the database, transaction commands, operations related to the rule management (e. g., activate, deactivate), or the invocation of (external) methods. It is performed only if the condition holds.

For some applications it is useful to delay the evaluation of a triggered rule's condition or the execution of its action until the end of the transaction, for example, when evaluating a consistency constraint that involves several updates. In other cases it may be useful to evaluate a triggered rule's condition or execute its action in a separate transaction. These possibilities resulted in the notion of *coupling modes*[18]. Coupling modes can specify the transactional relationship between a rule's triggering event, the evaluation of its condition and the execution of its action. Coupling modes can be defined when specifying ECA rules. The originally proposed coupling modes include:

– **Immediate:** The condition (or action) is evaluated instantly (synchronously) suspending the execution of the current transaction.
– **Deferred:** Condition evaluation (or action execution) takes place just before the transaction commits or rolls back.
– **Detached:** There are two possible ways to evaluate a condition/action in detached mode. Evaluating it *causally independent* means that the condition/action is evaluated completely asynchronously: a separate transaction is started. *Causally dependent* evaluation also starts a separate transaction, but not before the original transaction has committed or rolled back.

Most active database projects defined their own rule specification language but they share most of the features presented above. A small subset has become part of the SQL standard and is available in commercial DBMSs.

A difficulty in using ECA rules is to predict how the defined rules will behave in all possible scenarios. For some active database rule languages it is possible to perform automatic static analysis on sets of rules to predict certain aspects of rule behavior (i. e., conflicts, termination, or confluence) [65, 2, 40, 7]. Rule analysis techniques are dependent on the semantics of rule execution which are addressed in Section 2.3.

Defining Aspects: The AspectJ system we refer to [41, 5] is an extension to the Java programming language, therefore both base applications and aspectual behavior used to extend them are implemented in Java. Aspectual behavior is applied using information from the weaving description, a precise quantification of when and how aspectual behavior has to be interwoven with the application. AspectJ follows a compiler-based approach, interweaving aspect code with the base application at compile-time.

In AspectJ, aspectual behavior and weaving descriptions are usually expressed side by side in syntactical constructs called *aspects*. An aspect looks much like an ordinary class in that it may define methods and attributes, inherit from other aspects (including abstract ones) or classes and implement interfaces. Apart from that, aspects may contain *pointcuts* and *advice* to define crosscutting behavior, of which AspectJ supports two kinds. Static crosscutting means adding functionality to existing classes, while dynamic crosscutting allows for executing additional functionality at join points.

In terms of Sec. 2.1, AspectJ's join point model is dynamic, join points being nodes in the run-time object call graph. Both coarse- and fine-grained join points are supported, e. g., calls to methods, read or write accesses to objects' attributes or exception handler executions.

AspectJ uses *pointcuts* to quantify aspect applications. Pointcuts are composed of several join points and, optionally, values from the context of these join points that are of interest to aspectual behavior. Due to the dynamic nature of join points, it is possible to take whole partitions of the object call graph into account for quantification of pointcuts. AspectJ's `cflow` (control flow) pointcuts, parameterized with another pointcut, match whenever join points in the control flow of that parameter are met.

Aspectual behavior is defined in units of Java code called *advice.* Advice can be applied `before`, `after`, or `around` pointcuts, the former two plainly meaning that the advice code is executed before or after the code corresponding to a matched pointcut. Around advice are a more powerful instrument, allowing for modifications of the intended original control flow in that they completely *wrap* the code associated with the pointcut. A special statement, `proceed()`, can be used to invoke the original functionality at any point in the advice code.

2.3 Run-Time Semantics

Active Databases: Rule execution semantics prescribe how an active system behaves once a set of rules has been defined. Rule execution behavior can be quite complex, but we restrict ourselves to describing only essential issues here. For a more detailed description see [63, 1].

All begins with event occurrences signaled by event sources that feed the complex event detector that selects and consumes these events according to the specification of consumption modes. In the case of rules that do not involve a complex event, this step is omitted and event occurrences directly trigger the corresponding rules.

Usually there are specific points in time at which rules may be processed during the execution of an active system. The *rule processing granularity* specifies how often these points occur. For example, the finest granularity is "always" which means that rules are processed as soon as any rule's triggering event occurs. If we consider the database context, rules may be processed after the occurrence of database operations (small-granularity), data manipulation statements (medium-granularity), or transactions (coarse-granularity).

At granularity cycles and only if rules were triggered, the *rule processing algorithm* is invoked. If more than one rule was triggered, it may be necessary to select one after the other from this set. This process of *rule selection* is known as *conflict resolution*, where basically three strategies can be applied: (a) one rule is selected from the fireable pool, after rule execution the set of fireable rules is determined again, (b) sequential execution of all rules in an evaluation cycle, and (c) parallel execution of all rules in an evaluation cycle.

After rules are selected, their corresponding conditions are evaluated. The evaluation of the condition is performed according to the specification of transaction dependencies (coupling modes). If a condition evaluates to true, then the action associated with this rule must be performed. Actions can be any sequence of operations on or outside of a database. These operations can refer to attributes of the triggering event. Transaction dependencies are considered here too. This ends with a single execution of the rule processing algorithm. Additionally, rules can be activated and deactivated at run-time.

Aspect-Oriented Programming: Basically, all existing AOP systems with static weaving follow the same approach, as outlined in Sec. 2.1. Code for invoking aspectual behavior is interwoven with the base application's code. Thus, what happens at run-time in an application decorated with statically interwoven aspects is not greatly different from what happens in an application where the aspectual behavior is "hard-wired" into the code—in the end, all behavior that was cleanly modularized in aspects is again tangled with and scattered throughout the application code.

Whenever execution reaches a point where aspectual behavior was inserted by the aspect weaver, that functionality is invoked. The special case of more than one aspect applying to a single location in the code is handled as follows. Around advice are always executed first, followed by before advice, the actual base behavior, after advice and, finally, those parts of around advice that come after an eventual `proceed()` statement. If more than one advice of a kind are present, aspect domination and inheritance relationships are used to determine an order in which the advice have to be applied. This order is well-defined [41].

2.4 Run-Time Environments

According to the specification of ECA rules and aspects there is the need to signal that an application must produce a primitive event, respectively that a program has reached a join point or that a pointcut matches. The techniques used for this purpose can be characterized as *invasive* or *non-invasive*.

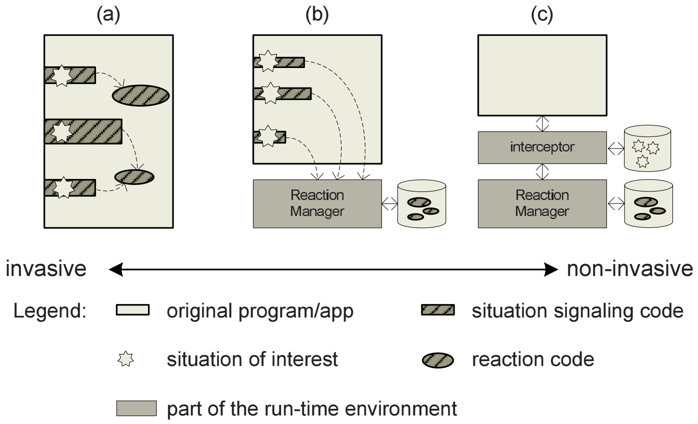

Fig. 2. Spectrum of code modification

In an invasive approach, at least the event signaling logic is mixed with the base application logic. In addition, reaction logic may also get mixed with base application logic. This mixed logic can take the form of instrumented source, byte, or native code, depending on whether the mixer is a pre-processor, a compiler, a byte-code toolkit, or even a just-in-time compiler. In Fig. 2a and b, invasive signaling is depicted by star-shaped wrappers. In Fig. 2a, in addition to the signaling code, the rest of the code that is necessary to execute the reactions is also included in the program. In Fig. 2b, signaling is also invasive but reaction code is separated from the signaling program and administered by a separate entity (which we call "reaction manager" to be domain-neutral).

Non-invasive techniques do not add any logic whatsoever to the base program. Rather the occurrences of situations of interest, e. g., certain operations in active DBMSs or join point execution in AOP systems, are somehow intercepted and, as a consequence, this situation is signaled. There are at least two approaches for doing this. If it is assumed that the application is developed with an interpreted programming language, then the language interpreter must be modified to signal situations of interest [20, 55]. A second possibility of non-invasive event detection is channel snooping [43]. In this approach the communication between client and server is intercepted and interpreted. This is useful when dealing with legacy systems that may not be modified.

Another approach in the implementation spectrum, which is not represented in the figure is one in which applications are programmed from the very beginning to generate events at interesting points. That is, developers are in charge of encoding the signaling of the happenings of interest.

We will now outline various implementation approaches to event detection, signaling and reaction execution in active databases and aspect-oriented programming. Please note that we do not claim either list of systems to be complete.

Active Databases

Invasive Approaches: Most active database systems use variants of the invasive approach. A fundamental difference is whether method wrapping is done manually or by the rule processor, and whether all methods are wrapped or only those methods for which rules are known to exist.

If selective wrapping is done, only those methods are decorated with the signaling code for which it is known that the event is required [28, 21]. This has performance advantages since no events are produced unless they are consumed and the application code is not bloated with unnecessary decorations. The disadvantage of this approach, however, is its inflexibility since new rules requiring new events that are not already produced cannot be added without modifying the application and recompiling it.

The alternative is to wrap automatically every method with a before and an after modifier [8, 66]. The method invocation events are always signaled locally. A fast table lookup is performed and if a subscription to that event exists, it is forwarded, otherwise it is discarded. The benefit is flexibility, since new rules can be added without modifying and recompiling the application code. The cost of this flexibility is acceptable in a database environment where data accesses involve entries in a lock table and accesses to external storage.

A compromise solution was adopted in [12, 10]. It consists in distinguishing between passive objects, reactive objects, and notifiable objects. The interface for reactive objects is modified and only methods in reactive objects are wrapped.

Non-invasive Approaches: Non-invasive approaches are convenient whenever the signaling application is non-cooperative or may not be modified, for example, in the case of legacy systems.

Channel snooping was used in [43]. In this approach a listener is implemented that taps into the communication channel between the client and the server. Since every relevant database operation is transmitted to the server, it can be intercepted, parsed and interpreted. The listener can then signal the occurrence of an event. There are obvious limitations to this approach. For example, events can only be interpreted at the query language statement level, i. e., at medium granularity according to our previous classification. Of course, they may be integrated to the coarser transaction level.

Modification of the interpreter is a powerful non-invasive technique. It was used in [20]. The benefit is flexibility to add new rules. The price is the difficulty of modifying the interpreter (if this is at all possible) and the inherent inefficiency of interpreted languages.

Aspect-Oriented Programming

Invasive Approaches: In terms of Fig. 2, within this category, we distinguish approaches that statically weave reaction code to the points of interest (at the level of source, byte, or native code), and those that only weave signaling code, leaving the reaction to be dispatched at run-time. This corresponds to the distinction between Figures 2a and 2b. AOP approaches corresponding to Fig. 2a directly mix reaction code into the application code before run-time. Hence, there is actually no such thing as situation signaling code. Both Hyper/J [35] and AspectJ [5] fall in this category.

Another class of AOP implementations for Java like EAOP [24, 23, 25], JAC [54, 37] and the *second* generation of PROSE [56, 57] all follow basically the implementation approach that corresponds to Fig. 2b. All three examples modify application classes by inserting hooks and/or wrappers at join points, thereby making the AOP infrastructure environment aware of them.

In more detail, EAOP uses a preprocessor to modify the application's classes before compilation, adding hooks in all places that may be a join point. JAC modifies classes' bytecodes as they are loaded into the virtual machine, inserting hooks in a specified set of places. PROSE 2 uses a modified just-in-time compiler to insert code that checks for the presence of advice at every possible join point in a group of classes that is specified at startup time of the AOP engine. All three approaches do not allow for altering the set of join points that are taken into account at run-time; they either activate all possible join points or only some in a given set of classes. Both approaches are unsatisfying: unnecessarily activated join points lead to – possibly expensive – checking operations, while an unalterable set of activated join points reduces flexibility.

So, on the one hand, the first category of invasive approaches statically binds reaction and disallows for dynamically dispatching of advice. On the other hand, the second category facilitates dynamic reaction dispatch but statically binds the set of join points.

Non-invasive Approaches: Another class of dynamic AOP systems is represented by the *first* generation of PROSE [55]. PROSE 1 does not instrument any code; instead the VM intercepts the execution of join points and dispatches to advice code whenever a join point is encountered that is matched by a pointcut. PROSE 1 makes use of the Java VM's debugging facilities [39, 45] to generate events at join points during application execution and intercept execution there. Hence, it belongs to the category of systems represented by Fig. 2c. This implementation – and probably any other implementation treating events as first-class entities of the run-time environment – suffers from performance overheads introduced by event generation and processing logic. The running application has to be permanently monitored by the run-time environment. However, using infrastructures like JPDA [39] allows for dealing with join points in a very flexible way: they can be arbitrarily activated and deactivated.

Aspect-Aware Runtime Environments: Another category of systems promises to avoid the performance overhead problem of systems like PROSE 1 while preserving its flexibility with respect to extending and reducing the set of activated join points. In this category, we classify approaches that are based on aspect-aware run-time environments. All approaches mentioned so far are characterized by an *impedance mismatch* between their aspect-oriented programming model and their execution model, which is basically that of object-orientation. Truly AOP-supportive approaches have to address this impedance mismatch by being based on an execution model explicitly designed to support the requirements of aspect-oriented programming.

To be aspect-aware, an execution environment must have two prominent features. First, both join points and advice code have to be dynamically bound. In that respect, an aspect-aware execution environment combines the advantages of the two categories of invasive systems mentioned above. A consequence from this requirement is that such an environment allows for weaving and unweaving aspect implementations at run-time. To preserve type safety, this leads to the second consequence that an aspect-aware extension of the type system would be required. The second important feature of aspect-aware execution environments is that the environment must *itself* inherently support weaving.

AspectS [34, 6] and Steamloom [33] are first developments in this category. AspectS is an extension to the Squeak Smalltalk implementation [36, 58], and Steamloom is a Java VM extension based on IBM's Jikes Research Virtual Machine [38]. Both systems fall into the category represented by Fig. 2a, with the important feature that all instrumentation of application code is performed at run-time.

In AspectS, aspects are deployed and undeployed dynamically by sending appropriate messages to instances of aspect classes. For achieving this, AspectS makes use of the powerful meta-level of Smalltalk that provides access to constructs of the run-time system, such as classes' method tables, which are available as normal Smalltalk objects. As such they can be changed on the fly, for example by (un)deployment methods of aspect objects. In terms of our discussion, AspectS only *simulates* an aspect-aware execution model at the application level: the actual execution model is that of the underlying Smalltalk implementation, which is not inherently aspect-aware.

We believe that, by being a new programming paradigm, AOP should be supported directly by the execution model. This is the motivation for our ongoing work on run-time environments, a first outcome of which is the Steamloom VM extension [33]. Steamloom allows for (un)weaving aspects at run-time by recompiling instrumented byte-code fragments that contain calls to advice code at join points.

3 Towards a Reactive Functionality Service

3.1 On the Convergence of Active Databases and AOP

Both aspects and ECA rules improve the separation of concerns and allow the implementation of crosscutting concerns. The discussion so far, has shown that Aspect Oriented Programming and Active Database Functionality present striking similarities, not only in the basic paradigm of reacting to defined situations through the execution of code, but also in the invocation mechanisms and primitives used. In this section, we will summarize these commonalities and will briefly discuss, how we envisage them to converge to what we call reactive functionality. Table 1 summarizes and compares the features and corresponding terms.

Table 1. Terminology in context

	AOP	Active Databases
simple situation	join points	primitive events
complex situation	pointcuts	composite events
[precise] situation	if [AspectJ]	condition
reaction	advice	action
execution	immediate	coupling modes

As described in Sec. 2.2, we have shown that there exist direct correspondences between join points and primitive events, pointcuts and composite events, conditionals in AspectJ and conditions in ECA rules, and advice and actions. An event in an active system denotes the occurrence of a situation that may lead to the execution of an ECA rule's action. In turn, the occurrence of a join point denotes that program execution has reached some specified point at which the invocation of some additional functionality, the associated advice, becomes possible.

Similar to event occurrences, pointcuts carry also additional information such as type and additional information (parameters or context). For instance, pointcuts can be signaled whenever a method `m()` of a class `C` is entered or a member `x` of an instance of `T` is accessed. Under these circumstances, the type for these join points can be `METHOD_CALL` or `MEMBER_ACCESS` respectively. Additionally, the name of the method or variable in question, the execution stack, etc. can be seen as contextual information.

Complex pointcuts in AspectJ are composed of primitive pointcut designators that are combined with logic operators such as `&&` or `||`. The only parallel between this kind of composition and that of composite events (as mentioned in Section 2.2) is the *alternative* operator. *Sequences* and *closures* of pointcuts/events are not taken into account, so AspectJ and other existing AOP approaches do not facilitate the composition of pointcuts to form what can be called a *sequential pointcut.*

A sequential pointcut, built using the event sequence operator, is complementary to the AspectJ `cflow` pointcut designator [41] that is able to match pointcuts occurring in the control flow (call tree) of a method. A sequential pointcut can match pointcuts by taking into account the event history of an application execution [61]. Such sequential pointcuts have the advantage of enabling an aspect to react to far more complex situations.

The task of a condition is to check for circumstances that go beyond the reach of the event—imagine an aspect or a rule that is to become active on the first day of the month only. The AspectJ `if()` pointcut designator [5] represents an effort to enrich pointcuts by a conditional part. However, the `if()` condition may only embrace variables that are bound in the pointcut.

As far as execution modes, AOP so far has been concentrating on synchronous execution of aspects, which corresponds to immediate coupling in active databases. We have shown that asynchronous execution of aspects may be a useful execution mode and we postulate that it will be necessary as we move toward a reactive middleware infrastructure. Another point to be taken into account is the concept of *coupling modes.* The question to be asked here is if aspect functionality *really* has to be executed *synchronously.* The common logging aspect usually consists of a call to some code that outputs information to a stream. If this happens synchronously, as with existing AOP implementations, the application decorated with the logging aspect pauses for the amount of time needed for logging. Considering that the output arguments are passed by value, it is possible to actually process the output in a separate thread and asynchronously invoke some entry point to that thread, allowing the actual application to proceed meanwhile.

From the discussion in Sec. 2.3, it becomes clear that the run-time semantics of active databases are richer than the run-time semantics of current AOP approaches, more specifically AspectJ. There are no correspondences to concepts such as processing granularity and processing algorithm in AOP languages. While these concepts as they are found in active databases might be "domainspecific" for the area of databases, it is worthwhile to consider integrating similar concepts in the design space of AOP languages and systems as well.

In Sec. 2.4, we have shown how different run-time environments require a more or less invasive decoration of the code to allow for the introduction of aspects or ECA rules, respectively. We also showed how the different run-time environments support the binding of rules or aspects at different points in time, ranging from compile-time to run-time. It is interesting to observe that the various implementations of today's AOP run-time environments have their correspondence among the implementations of active database systems.

It is part of our ongoing research to determine to what extent the rich semantics of event algebras, event consumption modes, and coupling modes are needed in the AOP context. We believe that some form of algebra for the specification of complex situations will be useful in the AOP context. We further believe that AOP can benefit from previous experience with event consumption and asynchronous execution of reactions. On the other hand we are encouraged by the possibilities that AOP and truly dynamic languages afford us to avoid the impedance mismatch that results when implementing reactive middleware functionality with static programming languages. However, in order for this to come true, AOP language technology should evolve from mainly code transformation techniques toward true run-time support for aspects.

From the predictability point of view and as a consequence of extracting crosscutting concerns conflicts among them may occur. The AOP community has paid little attention to this topic that needs further research.

3.2 Toward a Marriage of Distributed Services and Aspects

The last decade in the development of many areas of computer science, including database technology, middleware, and programming languages, is characterized by moving from monolithic software systems toward systems that are dynamically composed of autonomous, loosely-coupled components or services, as a way to react to changes in the environment.

A cornerstone in this development are asynchronous communication mechanisms. In recent years, academia and industry have concentrated on such mechanisms because they support the interaction among loosely-coupled components. Loosely-coupled interactions promote easy integration of autonomous, heterogeneous components into complex applications enhancing application adaptability and scalability.

For instance, in CORBA event [51] and notification [52] services were introduced to provide a mechanism for asynchronous interaction between CORBA objects. In the Java platform the Java Message Service (JMS) [32] provides the ability to process asynchronous messages. JMS has been part of Java Enterprise Edition (J2EE) [60] since its origin but was incorporated as an integral part of the Enterprise Java Beans (EJB) component model only in the EJB 2.0 specification [19]. It includes a new bean type, known as message-driven bean (MDB), which acts as a message consumer providing asynchrony to EJB-based applications. This formally introduces the possibility to write pieces of software (beans) that react to the arrival of messages.

In the late-90s the active database community has moved toward a service-oriented approach with the purpose to fulfill the requirements of new applications. In particular, the unbundling approach [29] consists in decoupling the active part from DBMSs. Various projects like C^2offein [42], FRAMBOISE [27], and NODS [16] have followed this approach. However, they did not address adequately the characteristics of distributed systems, for example, the lack of a central clock and the impossibility to establish a total order of events [44]. This has an enormous impact on the composite event detector and also on the underlying event algebra. A service-oriented approach that supports distribution and heterogeneity has been proposed in [15].

In the aspect-oriented software development research several efforts are being made in developing techniques that postpone aspect weaving ever later in the life cycle of an application [55, 57, 54, 37, 48]. These approaches are valuable experiments to demonstrate the usefulness of late aspect weaving (be that load-time or run-time). However, they all build on top of existing object-oriented language implementation technology, hence, suffer from impedance mismatch. For instance, the work on JAC [54, 37] is based on framework, MOP, and byte-code modification [14] technology, while PROSE 1 [55, 57] builds on the Java Platform Debugger Architecture [39]. The work on Caesar [49] focuses on language design rather than language implementation.

First steps toward solving the impedance mismatch of AO systems are being made [33]. This encourages us to see a new kind of distributed service-oriented systems emerge. They will be the result of marrying aspect-aware run-time environments that support an AO model, as the one we envisaged to be the convergence of AOP and active database systems and which also properly supports distribution and heterogeneity as in [15].

The AOP and the active database approaches need to detect the situation of interest (e. g., the invocation of a method call and the signal of an event respectively). In middleware platforms, like CORBA, .NET and J2EE, service requests and method invocations can be intercepted allowing the possibility of detecting transparently these situations. Interceptors have the ability to enhance an existing framework and the support applications transparently [50, 62]. The container model approach [60, 22] handles this issue by generating code based on configuration files. In this way the application or component can add selected services to its functionality by specifying properties in a configuration file. DADO [64] proposes the use of adaptlets at points where the application interacts with the middleware supporting in this way programming crosscutting concerns in distributed and heterogeneous environments.

4 Summary and Ongoing Work

Aspect Oriented Programming and Active Database Functionality present striking similarities. As far as execution modes, AOP so far has been concentrating on synchronous execution of aspects, which corresponds to immediate coupling in active databases. We have shown that asynchronous execution of aspects may be a useful execution mode and we postulate that it will be necessary as we move toward a reactive middleware infrastructure.

As it was mentioned before, both aspects and ECA rules improve the separation of concerns and allow the implementation of crosscutting concerns. There are different alternatives to allow the introduction of aspects and ECA rules into a run-time environment depending on more or less invasive decoration of the code. It was also shown how the different run-time environments support the binding of rules or aspects at different points in time, varying from compile-time to run-time. It must be noticed that there is an impressive correspondence between the implementation of numerous active database systems and today's AOP run-time environments.

As part of our ongoing work we are developing AORTA (Aspect-Oriented Run-Time Architecture [4]), and a distributed active functionality service [15]. The proof of our hypothesis will come as we implement the distributed active functionality service on an aspect oriented platform.

Acknowledgments

The authors thank the anonymous reviewers for their helpful comments and suggestions.

References

1. ACT-NET Consortium. The Active Database Management System Manifesto: A Rulebase of ADBMS Features. *ACM SIGMOD Record*, 25(3):40–49, 1996.
2. A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of database production rules: termination, confluence, and observable determinism. In *Proc. of ACM SIGMOD*, pages 59–68, San Diego, California, June 1992.
3. E. Anwar, L. Maugis, and S. Chakravarthy. A new perspective on rule support for object-oriented databases. In *Proc. of ACM SIGMOD*, pages 99–108, Washington, D.C., May 1993. ACM Press.
4. AORTA Home Page. `http://www.st.informatik.tu-darmstadt.de/static/pages/projects/AORTA/AORTA.jsp`.
5. AspectJ Home Page. `http://aspectj.org/`.
6. AspectS Home Page. `http://www-ia.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/`.
7. Elena Baralis and Jennifer Widom. An Algebraic Approach to Static Analysis of Active Database Rules. *ACM Transactions on Database Systems*, 25(3):269–332, 2000.
8. H. Branding, A. P. Buchmann, T. Kudrass, and J. Zimmermann. Rules in an Open System: The REACH Rule System. In *Proc. of RIDS*, pages 111–126. Springer, 1993.
9. A. Buchmann. *Architecture of Active Database Systems*, chapter 2, pages 29–48. In Paton [53], 1999. In Paton, N. 1999.

10. S. Chakravarthy. SENTINEL: An Object-Oriented DBMS With Event-Based Rules. In *Proc. of ACM SIGMOD*, pages 572–575, Tucson, Arizona, USA, May 1997.
11. S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, and R. H. Badani. ECA Rule Integration into an OODBMS: Architecture and Implementation. In Philip S. Yu and Arbee L. P. Chen, editors, *Proc. of ICDE*, pages 341–348, Taipei, Taiwan, March 1995. IEEE Computer Society.
12. Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proc. of VLDB*, pages 606–617, Santiago de Chile, Chile, September 1994. Morgan Kaufmann.
13. S. Charkravarthy, V. Krishnaprasad, E. Anwar, and S. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *Proc. of VLDB*, pages 606–617, September 1994.
14. S. Chiba. Load-Time Structural Reflection in Java. In Elisa Bertino, editor, *Proc. of ECOOP*, volume 1850 of *LNCS*, pages 313–336. Springer, 2000.
15. M. Cilia. *An Active Functionality Service for Open Distributed Heterogeneous Environments*. Ph.D. Thesis, Department of Computer Science, Darmstadt University of Technology, Darmstadt, Germany, August 2002.
16. C. Collet. The NODS Project: Networked Open Database Services. In K. Dittrich et.al., editor, *Object and Databases 2000*, number 1944 in LNCS, pages 153–169. Springer, 2000.
17. U. Dayal, A. Buchmann, and D. McCarthy. Rules are Objects Too. In *Proc. of Intl. Workshop on Object-Oriented Database Systems*, volume 334 of *LNCS*, pages 129–143, Bad Muenster am Stein, Germany, September 1988. Springer-Verlag.
18. U. Dayal and et al. The HiPAC Project: Combining Active Databases and Timing Constraints. *ACM SIGMOD Record*, 17(1), March 1988.
19. L. DeMichiel, L.U. Yalcinalp, and S. Krishnan. Enterprise JavaBeans. Technical Report Version 2.0, Sun Microsystems, JavaSoftware, August 2001.
20. O. Díaz, N. W. Paton, and P. Gray. Rule management in object oriented databases: A uniform approach. In *Proc. of VLDB*, pages 317–326, Barcelona, Catalonia, Spain, September 1991. Morgan Kaufmann.
21. K. Dittrich, H. Fritschi, S. Gatziu, A. Geppert, and A. Vaduva. SAMOS in Hindsight: Experiences in Building an Active Object-Oriented DBMS. Technical Report 2000.05, Institut fuer Informatik, University of Zurich, 2000.
22. Microsoft .NET Home Page. `http://www.microsoft.com/net/`.
23. R. Douence, O. Motelet, and M. Südholt. A Formal Definition of Crosscuts. Technical Report 01/3/INFO, École des Mines de Nantes, 4 rue Alfred Kastler, 44307 Nantes cedex 3, France, 2001.
24. R. Douence and Mario Südholt. A Model and a Tool for Event-Based Aspect-Oriented Programming (EAOP). Technical Report 02/11/INFO, Ecole des Mines de Nantes, 2002.
25. EAOP Home Page. `http://www.emn.fr/x-info/eaop/`.
26. T. Elrad, R. Filman, and A. Bader. Aspect-oriented programming. *CACM*, 44(10):29–32, October 2001.
27. H. Fritschi, S. Gatziu, and K. Dittrich. FRAMBOISE - an Approach to Framework-based Active Data Management System Construction. In *Proceedings of CIKM'98*, pages 364–370, Maryland, November 1998.
28. S. Gatziu and K. R. Dittrich. Events in an Active Object-Oriented Database System. In *Proc. of RIDS*, Workshops in Computing, pages 23–29. Springer, 1993.
29. S. Gatziu, A. Koschel, G. v. Buetzingsloewen, and H. Fritschi. Unbundling Active Functionality. *ACM SIGMOD Record*, 27(1):35–40, March 1998.

30. N. Gehani, H. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases: Model & Implementation. In *Proc. of VLDB*, pages 327–338, August 1992.
31. E. N. Hanson. An Initial Report on The Design of Ariel: A DBMS With an Integrated Production Rule System. *SIGMOD Record*, 18(3):12–19, 1989.
32. M. Hapner, R. Burridge, and R. Sharma. Java Message Service. Specification Version 1.0.2, Sun Microsystems, JavaSoftware, November 1999.
33. M. Haupt, C. Bockisch, M. Mezini, and K. Ostermann. Towards Aspect-Aware Execution Models. `http://www.st.informatik.tu-darmstadt.de/database/publications/data/ObjectModelDraft.pdf?id=75`. Submitted for review.
34. R. Hirschfeld. Aspect-Oriented Programming with AspectS. `http://www-ia.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/Docs/AspectS_NODe02_Erfurt2_rev.pdf`.
35. HyperJ Home Page. `http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm`.
36. D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: the Story of Squeak, a Practical Smalltalk Written in Itself. In *Proc. of OOPSLA*, pages 318–326. ACM Press, 1997.
37. JAC Home Page. `http://jac.aopsys.com/`.
38. The Jikes Research Virtual Machine. `http://www-124.ibm.com/developerworks/oss/jikesrvm/`.
39. Java Platform Debugger Architecture Home Page. `http://java.sun.com/j2se/1.4.1/docs/guide/jpda/index.html`.
40. A. P. Karadimce and S. D. Urban. Conditional Term Rewriting as a Formal Basis for Active Database Rules. In *Proc. of RIDE'94*, pages 156–162, February 1994.
41. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. Lindskov Knudsen, editor, *Proc. of ECOOP*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.
42. A. Koschel and P. Lockemann. Distributed Events in Active Database Systems - Letting the Genie out of the Bottle. *Data & Knowledge Engineering*, 25(1-2):29–53, March 1998.
43. T. Kudrass, A. Loew, and A. Buchmann. Active Object-Relational Mediators. In *Proc. of CoopIS*, pages 228–239, Brussels, Belgium, September 1996.
44. C. Liebig, M. Cilia, and A. Buchmann. Event Composition in Time-dependent Distributed Systems. In *Proc. of CoopIS*, pages 70–78, September 1999.
45. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.
46. H. Masuhara and G. Kiczales. A Modeling Framework for Aspect-Oriented Mechanisms. In L. Cardelli, editor, *Proc. of ECOOP*. Springer, 2003.
47. M. Mezini and K. Ostermann. Modules for Crosscutting Models. In *Proceedings of the 8th International Conference on Reliable Software Technologies (Ada-Europe 2003)*, 2003.
48. M. Mezini and K. Ostermann. Object Creation Aspects with Flexible Aspect Deployment. `http://www.st.informatik.tu-darmstadt.de/staff/Ostermann/aosd02.pdf`.
49. M. Mezini and K. Ostermann. Conquering Aspects with Caesar. In *Proc. of AOSD*. ACM Press, 2003.
50. P. Narasimhan, L. Moser, and P. Melliar-Smith. Using Interceptors to Enhance CORBA. *IEEE Computer)*, 32(7):62–68, July 1999.
51. Object Management Group. Event Service Specification. Technical Report formal/97-12-11, Object Management Group (OMG), May 1997.

52. Object Management Group. CORBA Notification Service Specification. Technical Report telecom/98-06-15, Object Management Group (OMG), May 1998.
53. N. Paton, editor. *Active Rules in Database Systems.* Springer, 1999.
54. R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In *Proc. of Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION 2001)*, volume 2192 of *LNCS*, pages 1–24, Kyoto, Japan, September 2001. Springer.
55. A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect-Oriented Programming. In G. Kiczales, editor, *Proc. of AOSD.* ACM Press, 2002.
56. A. Popovici, T. Gross, and G. Alonso. Just-in-Time Aspects. In *Proc. of AOSD.* ACM Press, 2003.
57. PROSE Home Page. `http://prose.ethz.ch/`.
58. Squeak Home Page. `http://www.squeak.org/`.
59. M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On Rules, Procedures, Caching and Views in Data Base Systems. In H. Garcia-Molina and H. V. Jagadish, editors, *Proc. of ACM SIGMOD*, pages 281–290, Atlantic City, NJ, May 1990.
60. Sun Microsystems. Java 2 Enterprise Edition Platform Specification. Technical Report Version 1.3, Sun Microsystems, JavaSoftware, August 2001.
61. R. J. Walker and G. C. Murphy. Joinpoints as Ordered Events: Towards Applying Implicit Context to Aspect-Orientation. In *Proceedings for Advanced Separation of Concerns Workshop*, 2001.
62. N. Wang, K. Parameswaran, and D. Schmidt. The design and performance of metaprogramming mechanism for object request broker middleware. In *Proc. of COOTS'01*, January 2001.
63. J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, 1996.
64. E. Wohlstadter, S. Jackson, and P. Devanbu. DADO: Enhancing middleware to support cross-cutting features in distributed, heterogeneous systems. Technical report, Computer Science Dept., University of California at Davis, June 2003. `http://macbeth.cs.ucdavis.edu/dado.pdf`.
65. Y. Zhou and M. Hsu. A theory for rule triggering systems. In *Proc. of EDBT*, volume 416 of *LNCS*, pages 407–421, Venice, Italy, March 1990. Springer.
66. J. Zimmermann and A. Buchmann. *REACH*, chapter 14, pages 263–277. In Paton [53], 1999. In Paton, N. 1999.
67. D. Zimmer and R. Unland. On the Semantics of Complex Events in Active Database Management Systems. In *Proc. of ICDE*, pages 392–399, Sydney, Australia, March 1999. IEEE Computer Society Press.