TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

DISSERTATION

Managing Event Streams for Querying Complex Events

Ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften am Institut für

Software Technology and Interactive Systems
der Technischen Universität Wien

unter Anleitung von
Ao.Univ.Prof. Dr. Stefan Biffl
Ao.Univ.Prof. Dr. Christian Huemer
Dipl.-Ing. Dr. Alexander Schatten

durch

Dipl.-Ing. Mag. rer. soc. oec. Szabolcs ROZSNYAI

Godlewskig. 18/1
A-1220 Wien

Ort, Datum

Unterschrift

Kurzfassung

Um den heutigen Anforderungen einer agilen und adaptiven Geschäftswelt gerecht zu werden, haben sich Unternehmensprozesse zunehmend zu vernetzten und hochgradig komplexen Abläufen entwickelt, die parallel und mit wenig menschlicher Einflussnahme ausgeführt werden [55]. Organisationen müssen ihre Prozesse immer schneller den Marktgegebenheiten anpassen, um Veränderungen rascher zu adaptieren als ihre Konkurrenten. Daher verlangen zeitgemäSSe Geschäftsprozesse nach Agilität, Flexibilität und nach Serviceorientierung.

Die Eckpfeiler derartiger Geschäftsmodelle sind lose gekoppelte, verteilte und Service/Ereignis-getriebene (Service and Event Driven-Oriented) Systeme, die groSSe Mengen von Events auf verschiedenen Granularitätsebenen erzeugen. Events zu erfassen, zu verfolgen, Aggregierungen aus Eventketten zu bilden und Korrelierungen zwischen Events zu erkennen und zu verwalten stellen einen Untersuchungsinhalt dar, mit dem sich bereits verschiedene Forschungsgruppen beschäftigen [6][54][79].

Ein viel versprechender Lösungsansatz für diese Art von Problemen ist Complex Event Processing (CEP). Der Begriff Complex Event Processing wurde erstmals von David Luckham in seinem Buch The Power of Events [54] erwähnt und definiert eine Reihe von Technologien zur Verarbeitung groSSer Mengen von Ereignissen mit dem Nutzen diese zu überwachen, um Prozesse in Echtzeit zu optimieren und zu steuern. Ein CEP-System sollte in der Lage sein, Daten aus Ereignissen kontinuierlich in Echtzeit zu verarbeiten und zu integrieren ohne ETL-Batchprozesse zu verwenden, wie sie bei Data Warehouse Lösungen üblich sind.

Die Beiträge dieser Dissertation liegen im Forschungsbereich von Event Processing Systems mit einem speziellen Fokus auf CEP und Event-Processing/Query Languages.

Event-Based Component Model: Die hier präsentierte Arbeit stellt ein Event Component Model vor, das Event-Based Systems in einen breiteren Kontext von Ereignisverarbeitung setzt. Es definiert die Grenzen von solchen Systemen und beschreibt den Rahmen von Event-Driven Components. Im Besonderen entkoppelt das Modell, Event-Based Systems von der Kommunikationsinfrastruktur und bringt somit den Vorteil, dass die Fähigkeiten von der Ereignisverarbeitungsumgebung nicht durch die Kommunikationstechnologien beschränkt werden.

Event-Processing Models: Event Models haben einen starken Einfluss auf die Fähigkeiten von Event Processing Query Languages, auf die Flexibilität und die Usability von Event-Based Systems. In dieser Arbeit werden daher unterschiedliche Arten von Event Model Konzepten vorgestellt, diskutiert und evaluiert. Im speziellen wird in dieser Dissertation auf Event-Driven Sense and Respond Rules eingegangen, mit deren Hilfe man in der Lage ist Event-Action Entscheidungsbäume innerhalb des Ereignisverabeitungsmodells zu modellieren.

SARI-SQL Query Language: Der Hauptbeitrag dieser Arbeit ist die Vorstellung der Syntax und der Semantik sowie die Evaluierung der Event-Query Language SARI-SQL und ihrer Subsprache EAExpression. SARI-SQL ist der Gruppe der Domain-Specific Languages zuzuordnen und ist daher in der Lage die speziellen Anforderungen und Charakteristika von Events und deren Beziehungen zu erfüllen. Die Abfragesprache zum Abrufen von Real-Time Events und zum Erstellen von Verbindungen mit historischen Events, Metrics und Scores ist im Gegensatz zum Event Clouds Indizierungsansatz [76][73][90] eine formal strukturierte Lösung, die ANSI-SQL erweitert. SARI-SQL erstellt eine Abstraktion des Event-Typen Modells durch die Kapselung der internen Datenrepräsentierung. Der Benutzer dieser Sprache kann sich daher direkt auf die Formulierung der gewünschten Ergebnisse konzentrieren anstatt sich vor allem damit zu beschäftigen, die Dinge zum Laufen zu bringen. Dadurch wird es Domänen-Experten möglich, einfach und schnell Erkenntnisse aufgrund der Abstraktionsebene des spezifischen Problems zu gewinnen.

Die Ergebnisse dieser Dissertation stellt der Forschungsgemeinschaft und interessierten Gruppen ein generisches Komponentenmodell für Event-Based Systems bereit. Das vorgestellte Modell ist erfolgreich mit der Implementierung der Event-Base evaluiert worden, wobei SARI-SQL ein integraler Bestandteil der Event-Base ist und dessen Implementierung eine groSSe Herausforderung im Sinne einer performanten Abfrage von Events und der Korrelierungen war. Ein spezieller Fokus in dieser Arbeit ist auf die Sauberkeit und die Ausdrucksfähigkeit des Sprachdesigns gelegt worden, damit alle Event-zugehörigen Entitäten berücksichtigt werden können. Des Weiteren ist ein Schwerpunkt auf ein effizientes Design der Query Vorbereitung und der Ausführungs- und Auswertungsarchitektur gelegt worden mit deren Hilfe man verschiedene Query Optimizer Strategien einbinden kann. Mit der - in dieser Arbeit vorgestellten - Optimizer Strategie korreliert die Performance von Queries auf sogenannten Single-Value Types direkt mit der Performance des darunterliegenden RDBMS.

Zukünftige Forschungsarbeiten basierend auf SARI-SQL umfassen die Verbesserung der Optimierungsstrategien im Zusammenhang mit Nested-Attribute Types von Events. Dies inkludiert Query Analyseprozeduren und Strategien für Ausführungspläne damit die Anzahl von In-Memory Post-Evaluierungsoperationen reduziert wird und somit die Performance steigt.

Die hier präsentierte Arbeit ist Teil von längerfristigen Forschungsbemühungen mit dem Ziel ein umfassendes Set von Analysewerkzeugen zu designen und zu entwickeln. Diese Werkzeuge sollen den Benutzern erlauben ein groSSes Repository von Real-Time und historischen Events von verschiedenen Quellen abfragen und analysieren zu können. Zusätzlich ist das Ziel eine Konsolidierung und die Schaffung eines vereinheitlichten Event Models für Event-Based Systems, die eine breite Palette von Ereignisverarbeitenden Systemen unterstützt. Im Forschungsfokus steht dabei auch der Aspekt der Visualisierung von Events, deren temporaler Ordnung, den Korrelationen zu anderen Events und Event-Clustern.

Abstract

Nowadays, business processes evolved to networked workflows that are complex and executed in parallel with little human involvement to meet the needs of today's agile and adaptive business [55]. Contemporary business requirements yaw for agility, flexibility and service orientation. A simplified summarization of this widely discussed and necessary business trend can be reduced to the demand, that today's businesses have to adapt their processes and organizations faster than their competitors. Business organizations that are able to handle critical business events faster than their competitors will end up us winners in today's globalized and fast business.

The pillars of such business models are loosely coupled, distributed and service- or event driven-oriented systems that generate huge amounts of events at various granularity levels. The lack of tracking those events and maintaining the causal relationships and traceability between those events, as well as aggregating them to high level events or correlating them, is a problem that is currently investigated by many research groups [6][54][79].

Event-based systems are increasingly gaining a widespread attention for such classes of problems, that require integration with loosely coupled and distributed systems for time-critical business solutions. The field of event-based or event-processing systems is a quite young area of research and is mainly influenced by the publish-subscribe paradigm and relational database and later on by Active- and Zerolatency data warehousing.

A promising solution for these problems is Complex Event Processing (CEP). The term of Complex Event Processing (CEP) was first introduced by David Luckham in his book The Power of Events [54] and defines a set of technologies to process large amounts of events, utilizing them to monitor, steer and optimize the business in real time. A CEP system continuously processes and integrates the data included in events without any batch processes for extracting and loading data from different sources and storing it to a data warehouse for further processing or analysis. CEP solutions capture events from different sources, with different time order and take events with various relationships between eachother into account.

The contributions of this dissertation are settled in the research area of event processing systems with a special focus on CEP and Event Processing- and Query Languages.

Event-Based Component Model: The presented work introduces an event component model that puts event-based systems into a broader context of event processing, defines the boundaries of such systems and the scope of event-driven components. In particular the introduced model decouples event-based system completely from the communication infrastructure, which offers the advantage that the capabilities of an event processing realm are not constrained by the underlying communication technology.

Event-Processing Models: Several new concepts of event models are introduced as their design strongly constraints the capabilities of event processing query languages. Further they have a major impact on the flexibility and usability of event-based systems. A special attention in this thesis was also set on event-driven sense and respond rules, which can be used to model trees with event actions within the event processing model.

Event-Base: This thesis introduces the design and the concepts of the Event-Base which is an extension of the event processing system SARI [81]. SARI allows to observe relevant business events to identify exceptional situations, indicates opportunities or problems combined with low latency times in decision making for supportive or counter measures. The Event-Base, on the other hand, provides an efficient up-to-date operational storage together with retrieval mechanisms for business events for analytical as well as operational purposes without the costly data staging processes known from established data warehousing solution.

SARI-SQL Query Language: The major contribution is the introduction of the syntax, semantics and the evaluation of the event query language SARI-SQL and its sub-language the EAExpression. SARI-SQL can be allocated to the group of domain-specific languages. The query language allows to retrieve near real-time events and create conjunctions with historical events, metrics and scores. Furthermore, it is in contrast to Event Clouds indexing approach [76][73][90] a formally structured solution that extends ANSI-SQL. SARI-SQL creates an abstraction of the event type model by encapsulating a lot of overhead and creating an abstraction layer over events and their internal data structures. The user of this language can concentrate on only expressing the required results instead of putting effort into making the "things run". As a consequence it allows domain experts to easily gain insights due to the level of abstraction of the specific problem domain. So for instance it allows retrieving events according to several constraints and access event correlations.

The results of this dissertation provide the research community as well as interested parties with a generic component model for event-based systems. The introduced model has been successfully evaluated through the implementation of the Event-Base. SARI-SQL is an integral part of the Event-Base and its implementation was a major challenge in terms retrieving events and their correlations in a reasonable time. The presented work set a special focus on a clean and expressive language design in order to encapsulate all the event-related entities. Furthermore an emphasis was set on an efficient design of the query preparation and evaluation architecture that allows attaching different query optimizer strategies. With the introduced optimizer strategies the performance of queries on single-value types (which applies in 80% of the cases) is directly correlating with the underlying RDBMS performance constraints and thus creates only a small overhead.

The future work on SARI-SQL includes efforts in optimizing the strategies of handling nested attribute types of events. This includes query analysis procedures and execution planning strategies in order to reduce the number of in-memory post-evaluation operations.

The presented work is part of a long-term research effort aiming at designing and developing a comprehensive event analysis toolset that allows users to query and analyze large repositories of real-time and historical events from various sources. In addition the goal is to consolidate and create a rich unified event model for event-based systems which can be supported by a wide range of event-based systems. A key focus of future research is also set on the aspect of the visualization of events with respect to their temporal occurrence, their correlation with other events, and event clusters.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

August 18, 2008

Acknowledgments

I would like to thank Josef Schiefer, Stefan Biffl and Alexander Schatten for the mentoring throughout the PhD thesis. Moreover I would like to thank my parents for all the support during my studies.

Contents

1 Introduction and Research Statement

Nowadays, business processes evolved to networked workflows that are complex and executed in parallel with little human involvement to meet the needs of today's agile and adaptive business environments [55]. Such contemporary business requirements yaw for agility, flexibility and service orientation. A simplified summarization of this widely discussed and necessary trend can be reduced to the demand that today's businesses have to adapt their processes and organizations faster than their competitors. Organizations that are able to handle critical events faster than their competitors will end up us winners in today's globalized and fast paced businesses.

The pillars of such agile models are loosely coupled, distributed and service- or event driven-oriented systems that generate huge amounts of events at various granularity levels. The lack of tracking those events and maintaining the causal relationships and traceability between those events, as well as aggregating them to high level events or correlating them, is a problem that is currently investigated by many research groups [6][54][79].

Event-based systems are increasingly gaining a widespread attention for classes of problems that require integration with loosely coupled and distributed systems for time-critical business solutions. The field of event-based or event-processing systems is a quite young area of research and is mainly influenced by the publish-subscribe paradigm, relational databases and later on by active- and zerolatency data warehousing.

A promising solution for these problems is complex event processing (CEP). The term of complex event processing was first introduced by David Luckham [54] and defines a set of technologies to process large amounts of events, utilizing them to monitor, steer and optimize the business in real time. A CEP system continuously processes and integrates the data included in events without any batch processes for extracting and loading data from different sources and storing it to a data warehouse for further processing or analysis. CEP solutions capture events from different sources, with different time order and take events with various relationships between each other into account.

This dissertation introduces the CEP system SARI [81] and its processing components. SARI is a solution that is capable of processing large amounts of events, providing facilities with the capability to monitor, steer and optimize business processes in real-time. The major contribution of this thesis is the introduction of the Event-Base, which extends SARI's event processing model with an efficient up-to-date operational storage, together with retrieval mechanisms for business events for analytical as well as operational purposes. The query language for retrieving near real-time events and create conjunctions with historical events, metrics and scores is SARI-SQL and is in contrast to Event Clouds indexing approach [76][73][90] a formally structured solution that extends ANSI-SQL. SARI-SQL can be allocated to the group of domain-specific languages and therefore it is capable to satisfy the special requirements and meet the characteristics of events and their relationships. SARI-SQL creates an abstraction of the event type model by encapsulating a lot of overhead and by creating an abstraction layer over events and their internal data structures. The user of this language can concentrate on only expressing the required results instead of putting effort into making the "things run". As a consequence it allows domain experts to easily gain insights due to the level of abstraction of the specific problem domain.

1.1 Dissertation Outline

The structure of this dissertation is organized in six parts:

Fundamentals. The first chapter provides the reader with essential background information on terminologies to gain a common vocabulary and understanding for the topics discussed throughout this dissertation.

Event-Based Messaging Middleware. This chapter discusses various middleware concepts that provide means to decouple communication parties in order to assure flexibility, scalability and fault tolerance. These concepts lay the essential foundations for event-based systems. Further, the chapter introduces a generic component model of an event-based system.

Emerging Event Processing Paradigms. This chapter discusses and compares the concepts of complex event processing and event stream processing. Furthermore, it describes their importance in low latency decision making and how these concepts can be utilized to detect and react on exceptional business situations in real-time.

Event-Base. This chapter introduces and discusses the components of the CEP system SARI and its extension the Event-Base. In addition it pays attention to special aspects of the event processing models such as event models or event correlations.

SARI-SQL Query Language. This section introduces the two domain-specific languages EAExpression and SARI-SQL which allow to define queries over streams of events. The languages are built upon the event management systems introduced in the Event-Base chapter.

Related Work and Comparison. The last chapter provides a comprehensive overview of various research efforts from the field of event-based systems. A special focus was set on their language capabilities especially in comparison with the solution introduced in this dissertation.

2 Fundamentals

The aim of this chapter is to provide the reader with the essential background on terminologies to gain a common vocabulary and understanding for the topics of the subsequent chapters. The reader might ask himself, why it is important to start with such basic topics like the clarification of the term event or communication and interaction models, although the main issue of this thesis is the query language SARI-SQL.

The research for this thesis has shown that the field of event-based systems is quite young and still in a consolidation phase. The result is that many terms and aspects of this field are ambiguous and depending on, from which field of research the reader comes from, different meanings can be easily associated to various topics.

For instance, the roots of event-based systems come basically from two different fields. The first one has grown out of append-only databases and can be seen in the world of triggers and zero/active data warehousing. The second root can be allocated to the development of middleware systems, the publish/subscribe paradigm and the various filtering and subscription techniques. Such subscription efforts formed the first rudimentary event processing engines including languages comparable to a certain extent to SARI-SQL. Each of those fields formed event-based systems with a partly different terminology and understanding. However, at the end of the day the solutions mostly provide the same capabilities. An ongoing debate that reflects this issue is currently about event stream processing and complex event processing where both provide comparable functionalities, but their roots and their event processing approaches are different. Another example is the view of event processing systems and their scope. Solutions coming from the publish/subscribe field tend to consider the communication infrastructure as an essential part of an event-based system as they usually aim for wide-scale event processing with the trade-off for expressive event processing languages.

The goal of the first couple of chapters is to discuss the ambiguous meaning of various topic in order to create a common ground of understanding, that is required to precede with the more sophisticated issues introduced in this thesis. Especially the frequently used term *event* in context of event processing systems is highly ambiguous. At this point of time there is no formal and generally accepted definition. Therefore, the section will give the reader a definition of events that is valid throughout this thesis.

2.1 Discussion about Events

2.1.1 Definition of Events

The term *event* is an often used synonym in computer sciences and it is mostly known by software developers as means to control and steer the program flow influenced by interrupts coming from other components - like in graphical user interfaces where user inputs trigger events to start corresponding processes.

The reader will encounter the term event many times throughout in this thesis and in literature related to the topics of event-based systems, complex event processing (CEP) and event stream processing (ESP). The problem with this term is actually that it has an ambiguous meaning. At this point of time there is no formal and generally accepted definition of the term event in context of event processing systems. Thus the definition of events is under an ongoing discussion in the event processing community. The definition of the scope and the domain of this term is important to get a common understanding about the characteristics it reflects. The goal of this section is to provide a broader introduction of this frequently used term in order to clarify it's meaning within the context of event processing.

The field of temporal databases, active- and zero-latency data warehousing can be seen as one of the roots of the field of event processing (CEP and ESP) and hence the terminology of the event processing domain was strongly influenced by these areas of work.

Jensen et al. [52] introduced in their work about temporal database concepts (*A glossary of temporal database concepts*), one of the earliest definitions of events in this context. They defined events as a single and closed occurrence of some happening at a specific point in time:

> *An event is an isolated instant in time. An event is said to occur at time t if it occurs at any time during the chronon represented by t.* [52]

A chronon is considered by the authors as a non-decomposable unit of time. They spin the event definition further by expanding the term event to intervals:

> *An interval is the time between two events. It may be represented by a set of contiguous chronon.* [52]

Another work from the active database management systems field [94] by Zimmer and Unland pinpoints the occurrence of an event to a specific point in time - similar to Jensen et al. [52]. An additional event characteristic, defined by Zimmer and Unland, is an atomic constraint around the events:

> *A primitive event is assumed to be instantaneous and atomic; i.e., it cannot be further dismantled and happens completely or not at all. It is bound to a specific point in time.* [94]

The work by Makkonen et al. about *Applying Semantic Classes in Event Detection and Tracking* [58] contains a comprehensive compilation of definitions of events related to the domains of *event detection and tracking* as well as *topic detection and tracking* (TDT). Although this work sets the term event in relation to real-world happenings, like political occurrences in a foreign country, it provides three event definitions that are directly reusable in the field of event processing. Basically the definitions provided by Makkonen et al. [58] and Yiming Yang el al. [91] are comparable to the definitions from Jensen et al. [52] and D. Zimmer and Unland [94] coming from the database area of work.

> *An event is something that happens at some specific time and place* [91].

> *An event is a specific thing that happens at a specific time and place along with all necessary preconditions and unavoidable consequences* [29].

However, the third definition by Makkonen et al. [58] provides the essences that an event might be composed and is capable of forking as a consequence to further events during time.

> *An event is a dynamic topic that evolves and might later fork into several distinct events.*[58]

David Luckham, the author of the book "Power Of Events" (covers the fundamentals of complex event processing) [54], maintains, together with the Gartner analyst Roy Schulte, a comprehensive glossary of CEP and ESP terminology with descriptions and definitions on the website *complexevents.com* [56].

David Luckham proposes following event definition in his online CEP glossary:

> *An object that represents, encodes or records an event, generally for the purpose of computer processing.* [56]

In addition the definition from David Luckhams book *Power of Events*:

> *An event is an object that is a record of an activity in a system. The event signifies the activity. An event may be related to other events.* [54]

Both definitions from Luckham [56][54] consider an event as a subject of interest and as a container for data processing.

Ludger Fiege presents a definition which is comparable to Luckhams understanding of events:

> *Any happening of interest that can be observed from within a computer is considered an event.*[41]

Since this thesis scope is not aiming at providing a universally valid and formal definition of events, this section only aims to provide a clarification of the term "event" valid and relevant within this thesis and applied to the CEP and ESP processing terminologies.

The following event definitions are derived from the previously discussed definitions and describe the term event in context of CEP and ESP applications:

> **Definition 1:** *Events are defined as observable actions or relevant state changes that can be absorbed by IT systems.* [41][53][54].
>
> **Definition 2:** *Events can be decomposed to several causally related events. Several events can be aggregated to a high level events.*
>
> **Definition 3:** *Events mark a specific point in time or in an aggregated form the timespan of an activity.*

A relevant state change is usually a subjective occurrence and so are events. In most cases events mark a specific point in time, however often several events can mark the starting and ending point of intervals which then can be treated as events again although they haven't happened "physically" in the real world.

An example would be the transport of goods, shown in Figure 2.1, from one place to another. A transport consists of a start event (*TransportStart*) and end event (*TransportEnd*). In between the timeframe of the occurrence of the two events *TransportStart* and *TransportEnd* there are events that are generated by a thermometer. The thermometer continuously tracks the temperature of the delivery goods of the transport. If the *TransportEnd* event has been captured, the sequence of the related events (correlated events) can be evaluated. If the thermometer events of the transport exceed a specific threshold, the whole transport and the related events can be aggregated to an event that can be called as *TransportFailed*. In literature these type of aggregated events are often referred to *composite events*. This small examples shows that events don't necessarily have an atomic characteristics.

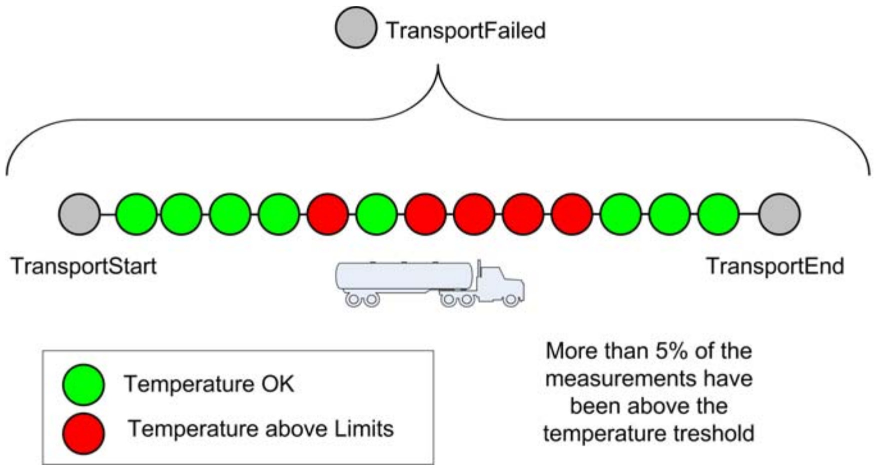


Figure 2.1: Transport Example

2.1.2 Representation of Events

The representation of information about the occurred activity is reflected through the attributes of events. The attributes are typed parameters and contain information about the specific action or state change. There are events that might have a high value to someone and, on the other hand, the same events don't necessarily have a meaning to other parties.

An important aspect of the events relevance to interested parties is their granularity level. If events are fine grained they do not necessarily deliver the right information to interested parties. People might only be interested on aggregated information or in calculated metrics for instance of a set of events. An illustrative example would be a HTTP client request and the corresponding response from a HTTP server. Looking at the OSI model where data flows across low level layers might not be a point of interest if the context of browsing web pages is taken into account. An event in the context of a HTTP request might be that the server application (a web-shop for instance) failed to process the order by the user. An event signalizing that some checksums failed in a lower layer is not a point of interest following the HTTP request context. This does not mean that the context cannot be accordingly expanded. If a HTTP request fails, events from the TCP/IP layer stack might be taken into account.

In the most common cases events are materialized or instantiated as messages, represented as semi-structured data like XML. XML was designed for the purpose of structuring data for electronic exchange which is the main reason why it has proven to be a good choice for middleware solutions.

2.1.3 Advanced Event Characteristics

In the following some event characteristics are reintroduced that were already accumulated and discussed in my previous diploma thesis *Efficient indexing and searching in correlated business event streams* [73].

Following David Luckham [54] events expose three key characteristics:

- ***Form:*** *An event is an object containing attributes or data components.*
- ***Significance:*** *An event signifies an activity.*
- ***Relativity:*** *An activity is related to other activities by time, causality and aggregation.*

Further on Luckham [54] defines three different types of relationships that events can have:

- **Time:** Relates events according to their temporal occurrence.
- **Cause:** Relates events according to their causal relationships. A causal relationship is given when an event A caused another event B in a direct or indirect way.
- **Aggregation:** Aggregates events to high level events based upon different criteria like time, causality or content patterns.

Following Luckham's [54] proposal that a timestamp defines the time relationship between events is not as easy as he says that you can determine by the time dimension that *event A happened before event B*. Leslie Lamport addresses this problem [53]:

> *In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relationship "happened before" is therefore only a partial ordering of the events in the system.* [53]

The happens-before relation → can be defined for following situations:

- If a and b are events in the same process, and a comes before b, then a → b.
- If a is an event sending a message from one process and b is an event receiving the message from another process then a → b.
- The happens-before relation → is transitive: a → b and b → c then a → c.
- If two events, a and b, are distinct, they are called concurrent which means that you can't say which event happened first. In other words it means that if the events a and b happen in different processes and they do not exchange messages, then a $\not\rightarrow$ b and b $\not\rightarrow$ a.

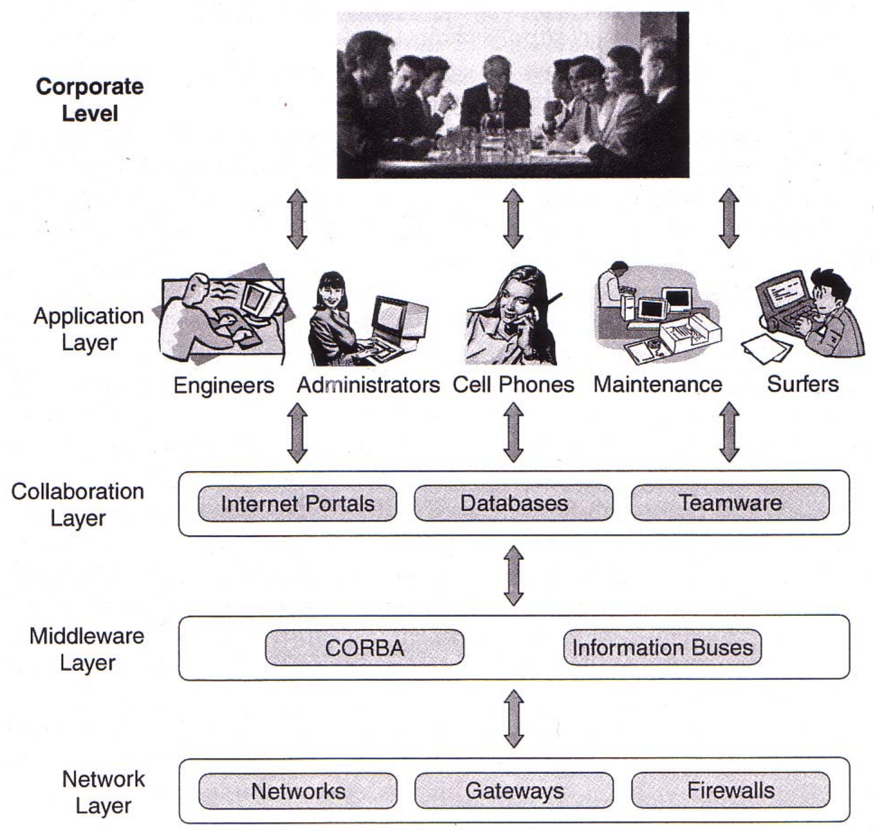


Figure 2.2: Enterprise Layers showing the flow of events according to [54]

As discussed in the diploma thesis [73] an additional important topic is addressed by Martin Fowler regarding timing issues [43]. Date and time information can come in different precision formats. You could get a date on a daily precision and you could get it on milliseconds precision level depending on your application. But as you work with events collected from different sources this could become a problem especially in distributed environments over the Internet where different time zones come in too. Furthermore, the question arises which event time is considered? The one where the event happened physically on a machine or when it has been processed somewhere else? This kind of questions have to be considered when building a system to process or collect events from different distributed sources. The aspect of different temporal characterizations have been discussed by Bruckner et al in [20].

In general a **causal relationship** determines that an event A caused another event B to fire. In the layered enterprise introduced by David Luckham [54] (shown in

Figure 2.2), events flow through several layers triggered by the user or another system at the top level. They are consequently transformed into lower level events down to the bottom causing other events to fire. This event flow is called *vertical causality* and tracks down how high level events beginning from the business level manifest in lower layers. This is important to understand how these events are decomposed the way down in order to be able to create meaningful aggregations out of the masses of low level events. On the other hand Luckham talks about *horizontal causality* that tracks the causal relationships between events at the same level. Causal relationships are transitive and asymmetric and can be represented as directed acyclic graphs.

> *Recognizing or detecting a significant group of lower-level events from among all the enterprise event traffic, and creating a single event that summarizes in its data their significance, is called event **aggregation**.* [54]

Aggregating events is a difficult task as it needs a technology that can recognize patterns of events through different layers. But if such an aggregation facility is set up and running, it can be a powerful source of tracking down causalities between events.

2.2 Communication and Interaction

Following Ludger Fiege's [41] distinction of interaction models, that were derived from an "Evaluation of cooperation models for electronic business" [63], this section will summarize the characterization of event-based systems.

Fiege, Mühl und Buchmann evaluated cooperation models for electronic business [63] back in 2000. In their work they referred to second generation e-commerce systems. Second generation e-commerce systems are characterized by a service-centric design to support the flexible composition of services to support frequently changing processes. They address virtual enterprises of the information-driven Internet economy whereas rigid cooperation models would not scale up with the agile requirements. In their work they state that event-based cooperation models would satisfy the flexible business requirements for such applications. Further they discuss several aspects if QoS parameters.

Virtual enterprises are an alliance, of possibly world-wide distributed companies, temporarily linked together, sharing their core competencies to exploit business opportunities [63]. This type of business model is usually characterized by short-term cooperations where every sub-service deliverer is concentrating on it's core competencies and after the business opportunity has been finished, reconfigurations of the business operations are required. *The success of a virtual enterprise largely depends on the efficiency to adapt the core services to changing demands of this environment* [63].

Although virtual enterprises do not exist today in such extensions, service centric IT infrastructure became one of the key points on the agenda in todays enterprises. Service-oriented architecture (SOA) is one of the most frequently used and misused terms. Basically, SOA is covering the topic of service centric, easily reconfigurable processes and transparently distributed IT infrastructure.

The infrastructure for enabling a flexible reconfiguration of services in order to support the implementation of frequently changing processes and organizations is a key success factor. An asynchronous way of delivering messages in distributed heterogeneous environments created a reliable and transparent solution of exchanging data between peers. The asynchronous messaging communication paradigm abstracts the underlying network complexities, hides the problems of different heterogeneous partners, allows loose coupling of peers, scales well with growing demands and brings the flexibility to meet today's requirements of agile organizations.

2.2.1 Basic Communication Models

Before getting into the details of interaction models there are six different communication models that should be noted:

- **Transient communication:** A transient communication is characterized by the elusive style of storing messages exchanged between a receiver and a sender. A message is only stored or held as long as the receiver and the sender are running. If one of the communication peers is not available anymore then the message is gone as well.

- **Persistent communication:** A persistent communication is the opposite to the transient communication as it stores exchanged messages in a persistent way. This guarantees, that in case the receiver or the sender stops working the message is not lost after a restart of the systems.

- **Asynchronous communication:** Asynchronous communication is characterized by having several parties that are exchanging messages. If a message sender is announcing a message, the message is delivered by the communication infrastructure in a transparent way to the receiver(s). The sender does not wait for any notification from the receivers and thus it can keep on working.

- **Synchronous communication:** Synchronous communication is the opposite of asynchronous communication as the sender is on hold until a notification is delivered by the receiver.

- **Broadcast:** Broadcasting messages is determined by a sender that is producing and sending out messages to multiple receivers. In contrast to other communication models the broadcast enforces that every available receivers gets a copy of a message.

Detailed information about communication paradigms, models and distributed systems can be found in [86].

2.2.2 Models of Interaction

The cooperation models according to [63] and [41] consist of the four interaction models Request/Reply, Anonymous Request/Reply, Callback and Event-Based. The cooperation model is distinguished by two attributes - the initiator and the addressee. The initiator can be either the consumer or the responder (e.g. provider) of an interaction. The initiator sends information about its own state, but it has no information about the recipients intention of what it is going to do with the state information. The second attribute is the addressee which determines if the addresses (e.g. the interacting peers) are known. In case there are not target addresses provided, a loosely coupling is given as the responder to service requests are chosen in an transparent way. By providing direct addressing a tight coupling is given, but the service functionality would be able to incorporate the information of the interaction partner into it's functionality.

		Initiator	
		Consumer	Provider
Addressee	Direct	*Request/Reply*	*Pt-to-Pt Messaging*
	Indirect	*Anonymous Request/Reply*	*Event-Based*

Figure 2.3: Cooperation Models according to [41]

2.2.2.1 Request/Reply

The Request/Reply interaction is usually related to classical client/server or command/control patterns. The initiator, a client system, requests a service, functionality or data from a provider. The request is processed and an answer delivered back to the request initiator. In this case the initiator is the consumer and the provider is the response system. Usually this type of interaction is done synchronously as the client awaits the response immediately. If request message is treated transient or persistent depends on the implementation of the communication infrastructure. Request/Reply interactions are well known as RPC or Java-RMI calls. Request/Reply creates a tight coupling between the communication parties.

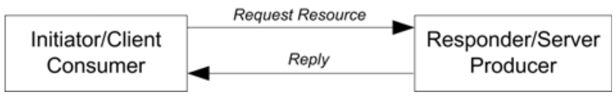


Figure 2.4: Request/Reply Interaction Model

2.2.2.2 Anonymous Request/Reply

From a behavior point of view, the anonymous Request/Reply interaction is comparable to the previously described Request/Reply interaction. The main difference is that the client, the consumer, is not addressing a specific provider with its request. The request is delivered, in a transparent way, to the corresponding producers. This type of request may result in one or more replies.

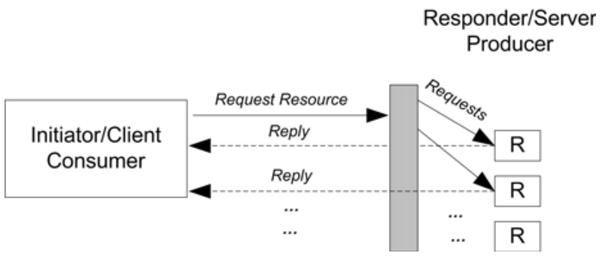


Figure 2.5: Anonymouse Request/Reply Interaction Model

2.2.2.3 Callback/Point-to-Point Messaging

The callback interaction model consists of a consumer and a producer whereas the producer is the initiator of a communication. The initiator is continuously producing and sending out notification about state changes - if they occur. The consumers, can announce their interest by registering to notification generating producers. The consumer can register itself to more than one producer. The producer is in charge of managing the registered consumers. This type of interaction is known as the observer pattern [45] and shows typical characteristics for an event-based communication. However, the producers and the consumers must manage the registration on both sides of established communications links. This solution leads to a tight coupling between the communication parties and thus significantly reduces the ability to adopt fast changing requirements.

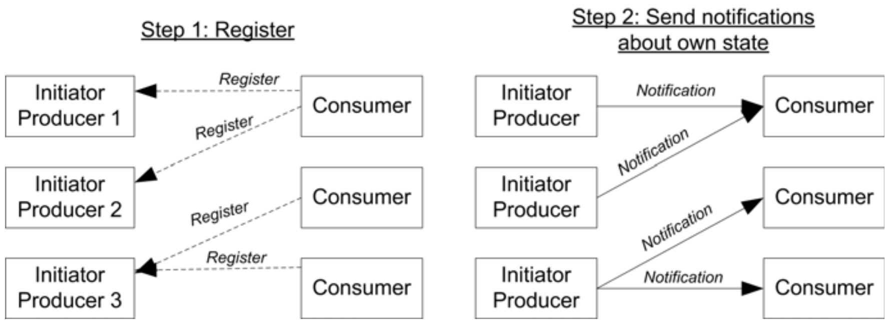


Figure 2.6: Callback/Point-to-Point Messaging Interaction Model

2.2.2.4 Event-Based Communication

The event-based communication model is similar to the callback model previously introduced. This type of model includes a producer and consumer while the producer is the initiator by sending notifications about its state. The main difference to the callback model is that the sender is not addressing a receiver explicitly. In the callback model the producer is maintaining a list of registered consumer that are updated about state changes. In the event-based model the producer *fires* notifications about state changes and an underlying communication infrastructure transparently distributes the notifications to the consumers. The key point here is that producers do not know anything about the consumers. Further, in the event-based communication model, the consumer is not registering itself directly to producing components. Consumers create subscriptions to specific events that they are interested in. Therefore, the consumer also does not know anything about the event-producing components. This leads to a high degree of loosely coupling between interacting components. Event notifications and their processing are done asynchronously. However, the main complexity instance in this communication model is the infrastructure that is coordinating the delivery of the messages. This infrastructure must meet several requirements that are especially addressed by message-oriented middleware (MOM) systems.

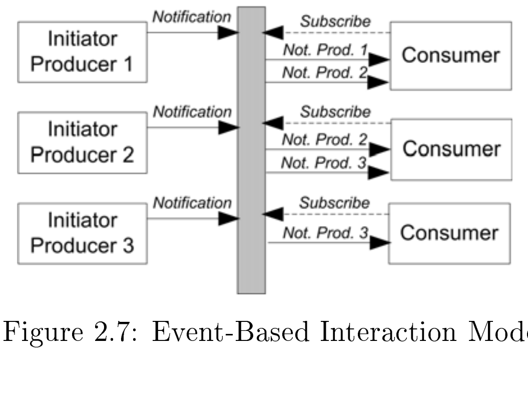


Figure 2.7: Event-Based Interaction Model

2.2.2.5 Communication Mediator

The infrastructure dealing with the management and coordination of the delivery of messages is often associated with the term *"event service"* [22][61] or *event notification service* [71][41] in the literature. An event (notification) service is often related to an event dispatcher that is responsible of coordinating the delivery of event notifications to the corresponding consumers. Consumers are subscribed to specific event notifications where the event service is managing those subscriptions.

There are several strategies of event (notification) service approaches.

- **Channels:** Consumer can subscribe themselves to specific channels that they have an interest in. The producers create notifications on specific channels.
- **Subjects:** Consumers can subscribe to several subjects. The event service forwards the notifications according to their subject classification to the corresponding authorities. Every notification emitted by a producer must contain a subject in such coordination environments.
- **Patterns:** There are more sophisticated approaches for event subscriptions like introduced by JEDI [34] where subscriptions can be expressed as event patterns.

At this point the reader should note that the term *event services* or *event notification services* will not be associated with the same meaning like described in the preceding paragraphs in order to be consistent with literature from this field. In the subsequent sections, after an overview was given and the prerequisites were discussed, this thesis will introduce an event-based architecture that is slightly different to already introduced concepts. The term "event (notification) services" is not associated with the same meaning. Event (notification) services are labeled with the term communication infrastructure in our proposed architecture.

2.3 Summary

This chapter outlined the basic terminologies that are required to understand the domain of event processing systems. In particular the term event was under a strong discussion as there is no formal definition of events available, in context of event processing, at this time. Therefore several works were introduced to clarify the scope and the domain of events. In addition a definition of events was given, that defines them as actions or state changes, whereas events can be decomposed, aggregated, can define a timespan and share relationships with each other.

Several prerequisites of communication models were introduced ranging from basics like asynchronous/synchronous communication to patterns and models, like callback messaging and event-based communication. This was the first occurrence where event-based communication was described. The key here is that communication models were developed to allow a loosely coupling between communication peers through a mediator that is in between them.

3 Event-Based Messaging Middleware

Messaging middleware systems play an important role in event-based systems as they provide means to decouple communication parties in order to assure flexibility, scalability and fault tolerance. Therefore, a section of this chapter will introduce important concepts for messaging middleware related to event-based systems.

The publish/subscribe paradigm has crystallized out to be a good choice for supporting event-driven solutions as they are loosely coupled, asynchronous and easily integratable into heterogeneous environments. Therefore a section in this chapter pays a strong attention to the publish/subscribe paradigm. Especially various filtering techniques are discussed as the content-based filtering technique contains elements that are the predecessor of what is called event-stream processing.

Service-oriented and event-driven architectures have gained a lot of attention both from technical point of view as well as from the marketing side. Especially SOA was hyped quite heavily in the past years from different analysts and praised to be an architecture that is going to replace tightly coupled and heavy to maintain monolithically designed applications. A section regarding architectural topics defines a distinction and presents a comparison of these two complementary concepts.

The component model of an event-based system is introduced and explained with detailed illustrations. It puts event-based systems into a broader context of event processing, defines the boundaries of such systems and defines the scope of each components. Event-stream processing and complex event processing will be defined and compared in a dedicated section as they might create confusion due to todays efforts by marketeers of various software vendors and business analysts that want to create a branding for this field.

At the end of this chapter the reader should have gained enough background knowledge to understand the building blocks of event-based systems that will be discussed in the next chapters and are the conceptual basis of the Event-Base.

Client/server and request/reply application architectures, where system components access remote infrastructure and functionality to accomplish their own tasks, have led to the requirement for decoupling the communication parties in order to assure flexibility, scalability, and fault tolerance in the communication infrastructure. An additional middleware layer helps to hide the heterogeneity of the underlying platforms and improves transparency. The possibility to exchange messages in distributed heterogeneous environments in an asynchronous persistent way [86] enables a reliable and transparent solution of exchanging data between peers. The message-oriented middleware paradigm has become one of the communication pillars of today's enterprise system and in particular for event-based systems.

The demand for the above described requirements can be found in daily business-scenarios. Many organizations invested a vast amount of money during time into their IT systems and applications creating a number of legacy components. For instance when companies are facing mergers or acquisitions, the problem of aligning their IT systems with the newly added organizations arises. Different departments might provide services or processes supported by various IT systems that need to be reorganized. In these scenarios it is important to integrate and reorganize the IT systems with low efforts which is in practice a difficult tasks due to the heterogeneous or legacy nature of such systems. Building new systems from scratch is rarely an alternative as it consumes too much time and reduce the time to market. Further developing applications or customizing package based solutions is a costly undertaking.

Event-based system architectures are distributed software components containing two interacting parties - an event producer and an event consumer. The communication model of event-based systems rely on asynchronous interconnections between the components [61]. An event producer is creating event notifications that are forwarded through a communication infrastructure to event consumers in an asynchronous way. One or more consumers can signalize their interest in specific events. The announcement of an interest establishes an anonymous communication link between independent components without making a central control authority necessary. The consumers are components that react on received events. This communication architecture allows event-based systems to enable a dynamic, fast and relatively easy reconfiguration of interacting components.

Ludger Fiege points out in his dissertation [41] that, although the term event-based system is widely used in computer sciences, there are quite a lot of ambiguous definitions and terminologies. Therefore, this section aims to give the reader the necessary background information on communication, in particular messaging paradigms that are prerequisites for understanding event-based systems and architecture concepts.

3.1 Message-oriented Middleware - MOM

As mentioned in the introduction of this chapter, client/server and request/reply communication patterns do not satisfy the requirements of decoupling communication parties in order assure flexibility, scalability and fault tolerance in communication infrastructures.

Client/server architectures are easier to build and maintain as their interaction complexity is relatively low in most cases and thus are easier to handle by software developers and designers. When trying to design and build distributed systems to meet the above requirements it becomes a difficult tasks as there are several points of failures which themselves are difficult to test and debug.

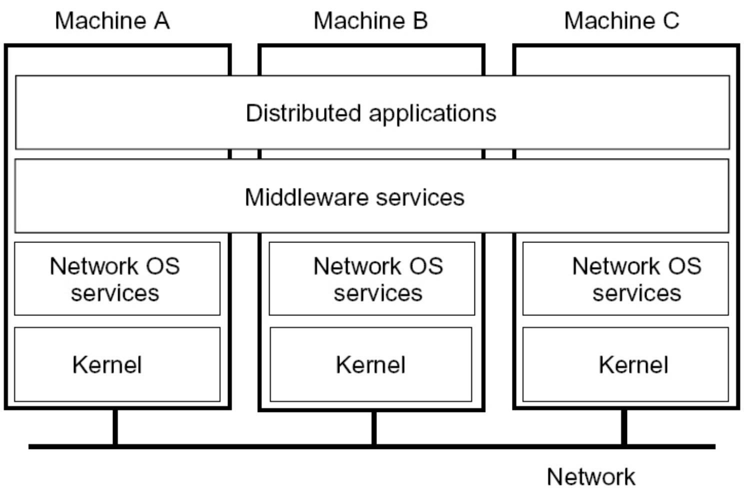


Figure 3.1: Illustration of a Middelware [86]

Therefore an additional middleware layer has been introduced to improve transparency and to hide the heterogeneity of the underlying system platforms. Figure 3.1 shows an illustration of a typical middleware [86] where different physically separated machines share resources. A middleware layer significantly reduces complexity and simplifies the development and integration of systems or applications into a distributed environment. Further, it allows a fast adoption of new requirements and preserves investments in legacy applications that can be reused as components for instance in new solutions to meet upcoming requirements. As shown in figure 3.1 each of the machines A, B and C can have their own, to each other different, operating systems and network operating systems (NOS). Almost every operating system provides a wide set of tools to manage local resources over the network. These tools are often associated with the term of a network operating system which can not provide a coherent view over various systems. The kernel, or the operating system in common, is only responsible for managing the local resources of one machine. Using the network operating systems for solving problems of distributed systems is highly time consuming, error prone and thus an expensive undertaking. In heterogeneous application landscapes it takes great efforts to develop and maintain applications that provide truly distributed functionalities. The heterogeneity makes it difficult to create an interoperability that satisfies the requirements of distributed communication in terms of flexibility, scalability and fault tolerance.

In literature, there are several definitions of middleware systems. Sternberger created in his diploma thesis [83] a comprehensive compilation of definitions about middleware. Sternberger uses a workaround for providing a definition of middleware by creating a collection of definitions found in literature. The compilation of definitions shows that there is no generally valid definition of middleware systems. This is because of the nature of middleware systems that have a wide set of applications and differentiates itself by them.

However, in the following a definition of middleware systems is valid throughout this thesis and describes the characteristics of middleware systems relevant for event-based systems.

> *Middleware can be defined as a mediator, between distributed and possibly heterogeneous software components, that is managing the communication by creating an abstraction layer to hide the interacting platforms.*

3.1.1 Types of Middleware

Tanenbaum's book on distributed systems [86] is a comprehensive work covering different topics and aspects of distributed systems including middleware. This sections gives the reader a summary of various types of middleware solutions. More on the topic of distributed systems and early middleware solutions can be found in [32] and [86].

- **Remote Procedure Calls (RPC):** One of the earliest approaches towards a middleware are Remote Procedure Calls (RPC). RPCs allow to call functions, including passing over parameters to remote machines. The major enhancement with RPCs was that they were capable of hiding the physical location of the remote machine. The RPC server provides an interface towards its clients and the server can participate as a client itself towards other RPC servers as well.

RPC services are usually implemented following the request/reply paradigm with the drawback of synchronous communication.

- **File-Based:** File-based middleware models are characterized by providing a distributed and scalable file system. Plan 9 was the predecessor of such a distributed file system centric middleware. In Plan 9 every resource is organized and treaded as files which then can be shared by processes and the communication interaction is limited to file accesses. File-based middleware models are very popular in the Unix world.
- **Distributed Objects:** Middleware models based on distributed objects mostly rely on remote object invocations. The distributed interaction between the interacting peers are represented as accesses to objects on remote hosts. The remote hosts provide interfaces for those objects. This model is related to Object Request Brokers known from CORBA.
- **Service-Oriented:** Service-oriented models are related to service oriented architecture (SOA) where web services are one of the most prominent concepts. They are based on distributing and bundling services with the goal to replace tightly coupled and heavy to maintain monolithically designed applications.
- **Message-Oriented:** Message-oriented models allow different heterogeneous parties to interact with each other through a middelware layer that provides a common infrastructure for exchanging messages. This type of middleware is completely ignoring the contents of the messages as it is only responsible of providing the infrastructure to exchange them.

3.1.2 Middleware Requirements

The goal of middleware systems is to solve the problem of integrating components into a heterogeneous application landscape without developing new communication interfaces for each of the components and thus reducing the development time, complexity and error proneness.

Following [36] there are five requirements that a middleware systems must fulfill:

- **Network Communication:** Due to the nature of distributed systems applications, components can be split up to several physical machines. As the applications have to communicate to each other through networks, distributed systems should be built on top the transport layer of the OSI model. They should encapsulate the session and presentation layer of the OSI model and provide interfaces or service towards application developers.

- **Coordination:** Components are distributed over different hosts, which requires some sort of synchronization techniques. If a client requests a service, the server can acknowledge the request and keep on performing other operations. Later on time the server processes the client's request and syncs itself with client or requestor. This type of behavior is referred to asynchronous communication discussed in the Section 2.2.1 *Basic Communication Models*. Synchronization mechanisms should be provided by middleware systems and implemented at the session layer.

- **Reliability:** Reliability, in terms of fail-safeness, error detection and correction are requirements that goes on the account of performance. Therefore it needs a trade-off between reliability and performance [36]. Fail-safeness can be achieved by providing means of replicating components. Reliability, in terms of guaranteeing the communication delivery is required for middleware systems. Middleware systems should be capable to execute transactions with ACID characteristics.

- **Scalability:** Scalability is one of the major issues that IT systems have to deal with. With growing demands, for instance from the business side, the systems need to scale up accordingly. A way to achieve this is to create an abstraction of the access to hosts. By making the location of hosts transparent, clients can access components or services without minding their real physical location or their identity. This allows to easily apply load balancing or replication services to scale up with growing demands or to catch load spikes.

- **Heterogeneity:** The IT landscape of large IT organization consists usually a wide range of different system components. Most of that systems need to interact with each other. The problems lies often in the fact that they have grown historically and thus they form a heterogeneous ring of systems in terms of operating systems, programming languages that software was written with and hardware. Some of these components might be open software with a rich set of interfaces, but on the other hand there are old legacy systems with proprietary interfaces. The middleware's job is to provide an abstraction of the heterogeneity instead of leaving this task to software developers.

3.1.3 Message Oriented Middleware

Message-oriented middleware (MOM) systems are a special type of middleware systems satisfying the requirements of network communication, coordination, reliability, scalability and heterogeneity introduced in the Section 3.1.2. A MOM is as a mediator component between interacting components providing an infrastructure for exchanging messages. Following Tanenbaum and van Steen a MOM provides *an extensive support for persistent asynchronous communication* [86].

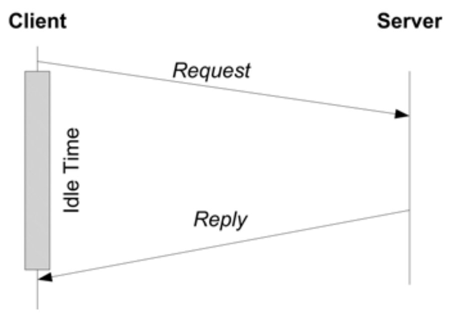


Figure 3.2: Client/Server communication paradigm idle time

The problem of the synchronous request/reply communication paradigm is basically that it requires both communication parties to be *on-line* or *alive* at the same time. This behavior causes a number of problems. Figure 3.2 shows the interaction model where a client sends a requests or a message to a server and the servers sends back an answer. The point in this figure is that the client is blocked or in an idle state until the server sends back its response. This type of synchronous communication leads to difficult error handling and creates a high connection overhead when trying to resolve failures. A simple error handling scenario for instance is when a server is not reacting on calls, the client has to retry to connect itself to the server until the request succeeded or the number of retries exceed a defined threshold.

One solution of this problem is to directly improve the synchronous communication paradigm by enhancing it with more sophisticated functionalities like load balancing or transactions. The other solution is to switch to asynchronous persistent communication paradigms. With such a communication paradigm the client sends a request and can keep on working until it receives a response. The client does not have to be running or ready for receiving the response as the response is stored until it is fetched. On the other side the request is stored until the server is alive or able to processes the request.

Asynchronous persistent communication paradigms can be realized with message queuing systems. Queuing systems allow components to communicate with each other through persistent queues. Persistent queues store messages in a robust way, that are fault tolerant and support transactions in order to catch exceptional situations and guarantee the integrity of messages. Message queues allow several components to access and read from queues. This enables failover through adding service replication and load balancing.

Message queues allow to model complex interaction patterns in completion to the Section 2.2.2. The queues can form a virtual connection between components that are called *channels*. Channels represent a communication connection between interacting components. The sender does not necessarily need to know who is the receiver of its messages and vice verse.

- **One-To-One:** One-To-One or Point-To-Point channels guarantee that a produced message is delivered to one and only one receiver. However the sender does not know to which receiver the message is delivered. This results in the situation that a producing component (e.g. sender) is issuing messages to the same channel, but the messages are not delivered to the same receivers.

- **One-To-Many:** One-To-Many channels allow to distribute one message to several receivers at the same time. In contrast to one-to-one channels this allows a producer to send one message and every listening component will get a copy of that message. Listening components are called subscribers. We will refer to this this type of channel as Publish/Subscribe later on in this chapter.

MOMs allow to define message queues for one or more interacting components. Often several components or applications share the same queue as they are interested in the same type of messages. By specifying such queues, components can interact with each other by sending messages to those queues. The messages are then enqueued and pending until another component (e.g. the receiver of the message) fetches the messages. This type of interaction is asynchronous and therefore the sender and receiver of the messages don't have to be in sync. Due to asynchronous communication MOMs provide persistence mechanisms for messages. MOMs don't care about the content of messages. They are only responsible of providing the necessary infrastructure for exchanging the messages and satisfying the requirements of middleware systems described in the previous Section 3.1.2.

The drawbacks of MOMs are related to asynchronous communication which can become a problem in time critical applications. They are not addressing security concepts especially in publish/subscribe systems.

Figure 3.3 shows an illustration of a message oriented middleware. There are four interacting components (A, B, C, D) and three message queues (X, Y, Z) in this illustration. The component A is sending messages to the queue X and the component B is sending messages to the queue Y. When the messages from the two components have been enqueued the component D is capable of fetching them from the queues X and Y. In this scenario, there is no designated producer or consumer for messages. At the first glance, the component D would qualify itself as a consumer. However, the component is not only fetching messages and consuming them, it is also producing

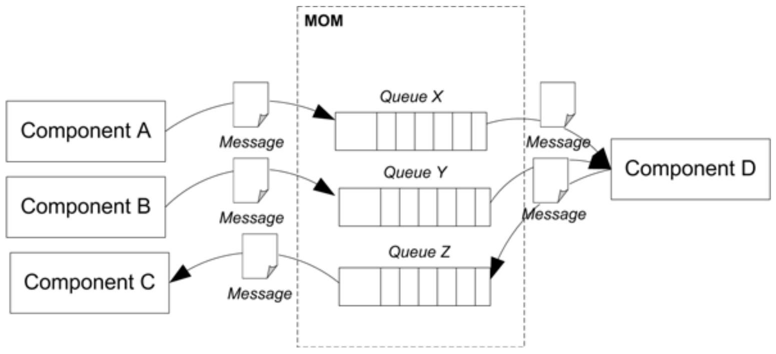


Figure 3.3: Illustration of a Messaging Middleware

messages and issuing them to the queue Z which is then consumed by the component C. The key in this illustration is that there are no designated producer and consumer roles. The MOM is encapsulating the interaction and providing message queues to the components. This enables loosely coupling and hides the heterogeneity of the peers.

Figure 3.3 shows a simple scenario of exchanging messages. The reader should bear in mind that MOMs have to perform tasks like routing messages to their destination. Sometimes routing of messages can become a complex task where messages are delivered through several queues until they reach their destination. Furthermore, MOMs have to deal with issues like data transformation in order to bridge the heterogeneity.

3.1.4 Terminology Clarifications

The following important terminology clarifications should be noted for the subsequent sections:

- **Event:** An event is an observable action or a relevant state change that can be absorbed by IT systems. In this case the IT system is the producer. Please refer to the Section 2.1 *Discussion About Events* for a detailed discussion .
- **Notification:** A notification represents an event, defines the encoding and the data structure.
- **Message:** The term message is related to concrete materialization of the notification. This can be for instance XML for representing a notification.

3.1.5 Types of Messages

A message is basically nothing else than a package containing data produced by a component and that is forwarded to a receiver. Typically, a message consists of a header and a body.

- The **header** contains data relevant for the MOM in order to deliver the message to it's destination. It can contain data about the sender, timestamp and priority.
- The **body** contains the "actual" content that is created by the producing component. Usually the body is of no interest to the MOM.

The MOM is not directly interested in the message body, but from the consumers point of view the messages can be differentiated by following types [62]:

- **Command Message:** Command messages are used to call functions or services of a system. For instance RPCs can be packed into messages to call the functionality.
- **Document Message:** Document messages can be used to transfer documents to other components for further processing.
- **Event Message:** Event messages are used to provide reliable notifications about events to other components.
- **Reply Message:** A reply message is issued to propagate back results to requested services or functions.
- **Message Sequence:** Message sequences are used to split up huge data amounts into several messages in order to be able to deliver them to its destination.

A lot of the message types introduced in this section and further described in [62] are related to the well-known design patterns [45]. Messaging can be used to implement design patterns such as the observer pattern for issuing event messages.

3.1.6 Publish/Subscribe Paradigm

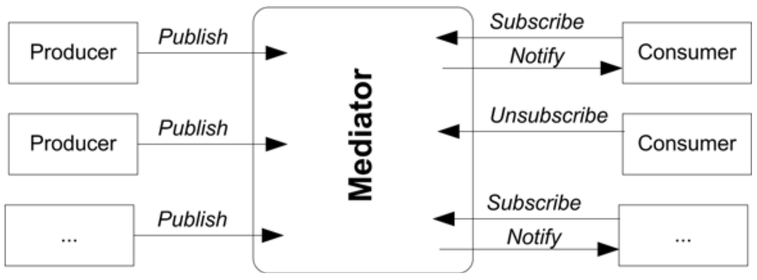


Figure 3.4: Publish/Subscribe Overview

The publish/subscribe Paradigm, or short pub/sub, is referring to many-to-many channels where a message producer is sending (e.g. publishing) messages through a channel. A channel is the simplest form of describing a virtual connection between peers. Channels are used in the beginning to introduce the basic concepts and later on more sophisticated concepts will be introduced.

The message consumer subscribes itself to a channel and receives the messages issued through the specific channel. There can be more than one consumer subscribed to one channel and thus one message is distributed to each of the subscribed parties. A message is considered as consumed when every subscribed party has received its copy of the message.

A pub/sub systems consists of following components:

- **Producer - Publisher:** This component is producing notifications or messages and distributes them to the mediator service which is basically a pub/sub middleware. In literature the term mediator is also called *event notification service* or *event service*.
- **Consumer - Subscriber:** This component is the consumer of notifications or messages. It signalizes its interest by subscribing to specific channels. A consumer can subscribe itself to more than one channel. The consumer component does not know anything about other consumers that are subscribed to the same channel for instance and thus receiving the same messages. Furthermore it is not explicitly aware of the implications that are cause by messages or notifications that are consumed by other components. A consumer can take in the role of a producer and act as a publisher.
- **Subscription:** A subscription is indicating the interest of a consumer/subscriber to a specific notification.
- **Mediator:** The mediator is the component that is responsible of delivering the right notifications to the corresponding consumers. This component is often referred to *notification service*.

One of the most important component of the pub/sub paradigm is the subscription mechanism that is managed by the mediator. Therefore some attention has to be spent on the various aspects and techniques of subscription types. The introduced concepts and techniques are used in event-based systems to provide an infrastructure for event-driven processing.

According to [37] the main strength of the publish/subscribe communication paradigm lies in the full decoupling of *time*, *space* and *synchronization* between the producers/publishers and consumers/subscribers:

- **Time:** The publish/subscribe communication paradigm is an asynchronous communication model which does not required the communication parties to be be on-line or alive at the same time while exchanging communication.
- **Space:** The consumers and producers don't have to know about each other and hence components in pub/sub messaging solutions are loosely coupled. The only interacting party for those two components is the mediator in this model.
- **Synchronization:** The consumers and producers are not blocked while producing or receiving notifications.

The characteristics described by [37] can be generalized to different types of messaging middleware (see Section 3.1.3).

3.1.6.0.1 Channel-Based

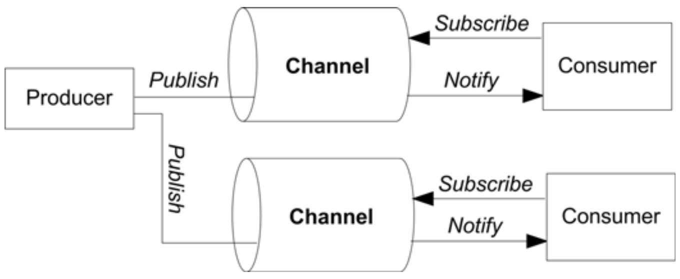


Figure 3.5: Channel-Based Subscription

As already mentioned in the beginning of this section and described in the section 3.1.3, channels are the basic concept of representing subscriptions. A channel is used by producers/publishers to distribute their notifications where the mediator redistributes them to the subscribed consumers. The drawback of this concepts is basically that there is no mechanism of filtering to different granularity levels which might result in an overhead of information that the consumer receives.

3.1.6.0.2 Topic- or Subject-Based

Topic-based subscriptions are a more advanced concept of controlling the distribution of notifications within a pub/sub middleware system. A topic can be seen as a label that is attached to notifications to classify their content. A notification can

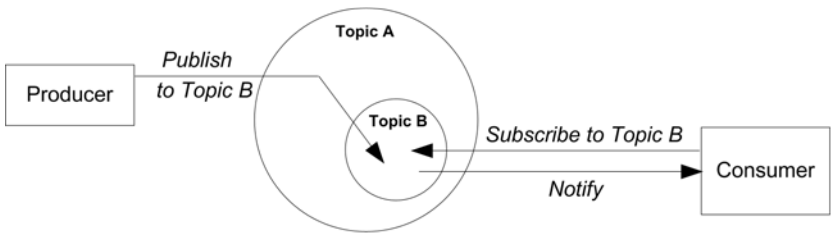


Figure 3.6: Topic-Based Subscription

now be categorized according to those *labels*. The consumers don't need to subscribe themselves directly to channels anymore, but to topics which are a more specific filter mechanism to control the potential flood of notifications.

An additional extension to these topics are hierarchies that are somewhat comparable to object oriented hierarchies. Hierarchies in topic-based subscription models don't have the characteristics of object-oriented concepts like inheritance and information hiding, but they allow to treat and access notification according to the set theory. However the notification selection/filtering is performed through matching strings [67] to the labels in ordered to divide the space of notifications.

Consider following topics $topicA \subset topicB$ where $topicB$ is a sub-topic of $topicA$. If a consumer is subscribed to $topicB$ it will receive all notifications that are addressed directly to $topicB$ but none of the notifications addressed to topic $topicA$. On the other hand a consumer that is subscribed to $topicA$ will also receive notifications that are directly labeled as $topicB$.

It is often possible to parametrize topic-based subscriptions with wildcards. Let's consider following example:

$$topicA \subset topicX \subset topicB$$
$$topicA \subset topicY \subset topicB$$

In this case there is a specialization that is the same at the lowest hierarchy levels. For instance if the consumer wants to subscribe itself to the lowest hierarchy level (e.g receive only $topicB$ notifications from every specialized branch) it can use wildcards:

$$topicA \subset * \subset topicB$$

Topic-based subscription are part of many commerical pub/sub messaging solutions. Their roots of topic-based approaches can be found in the context of group communication [18][19]

3.1.6.0.3 Type-Based

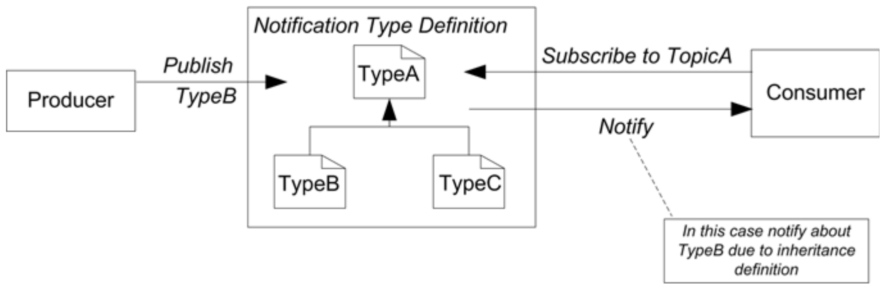

Notification Type Definition
Producer
Publish
TypeB
TypeA
TypeB
TypeC
Subscribe to TopicA
Consumer
Notify
In this case notify about TypeB due to inheritance definition

Figure 3.7: Type-Based Subscription

A type-based notification selection is relating to the classification of notifications according to notification types [15][40][37][67]. Notifications can be classified through different predefined types. The type definition is comparable to class definitions in object-oriented programming languages. Therefore type-based filter mechanisms support inheritance structures that allow to create generalization selections. The basic idea of this concepts is comparable to the topic-based approach with the main difference that in this case the notification service is not matching labels and their hierarchies against selection strings, but inferencing notifications against the predefined types and their inheritance hierarchies.

3.1.6.0.4 Content-Based

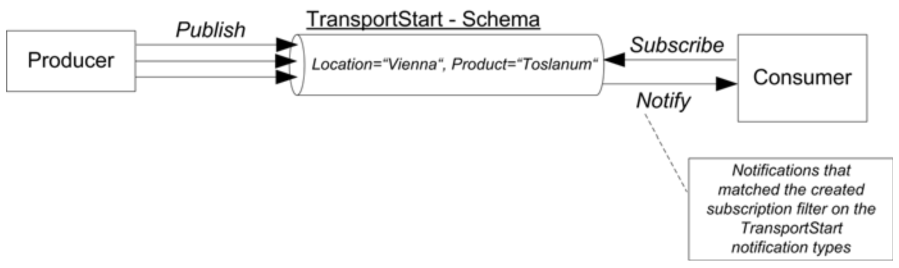

TransportStart - Schema
Producer
Publish
Location="Vienna", Product="Toslanum"
Subscribe
Consumer
Notify
Notifications that matched the created subscription filter on the TransportStart notification types

Figure 3.8: Content-Based Subscription

Due to the limited expressiveness of topic-based filter mechanisms, content-based subscriptions provide the ability to access the actual content of the notification itself [72][4]. That means that the consumer is capable of defining subscriptions, based on the internal data structures or attributes of the messages itself. This allows the mediator to provide a facility to create subscriptions, based on detailed notification-internal criteria instead of classification data that is attached to the notification like "labeling" them in topic-based approaches.

Content-based subscriptions provide means of specifying filters that can be defined through a filter language. Constraints can be created by defining key-value pairs, applying comparison operators and combining them arbitrarily with logical operators. By using these simple mechanism it is possible to form complex subscription patterns.

An important concept in content-based systems is the requirement of defining types of notifications and thus defining a detailed data structure of events. This data schema or data model encodes and represents events in form of notifications. This type of schema definition in combination with content-based filtering is the predecessor of event models we know from complex event processing. Event models for complex event-based systems will be discussed in detailed in the subsequent sections.

In topic-based subscription models the notifications are label with a topic which is a static schema and can't be used for filtering on finer granularity levels. Through matching notifications to defined schemas or types, the notification processing instance has information about the data structure of the notifications and thus it is possible to access their content.

Subscription languages are tightly coupled to the underlying event model and therefore their expressiveness is also strongly coupled to the underlying model. The authors of [24] are highlighting an interesting point coherence of expressiveness of subscriptions languages and scalability. The expressiveness of subscription languages goes to the account of the scalability of the pub/sub system due to additional overhead. A trade-off between the subscriptions languages and scalability has to be considered while designing such languages.

Bacon et al. [7] go one step further by introducing the concept of event correlations. The composite of these capabilities can be seen as the predecessor of event stream processing where you have a flow of messages that are matched by their content against specified patterns or expressions in the simplest case. Messages that are matching will be sorted out for further processing.

3.1.7 Event-Driven versus Service-Oriented Architecture

In the previous sections the reader was introduced to basic communication paradigms and infrastructural concepts. These concepts were explained with the meaning to provide essential background information on event-driven system, it's roots and their conceptual development. Before switching directly over to event-based systems this section aims to provide an understanding of event-driven architectures and it's characteristics especially in comparison to service-oriented architectures (SOA).

3.1.7.1 Service-Oriented Architecture

Service-oriented Architectures (SOA) have gained a wide spread attention both from technical point of view as well as from the marketing side. Without a doubt SOA was hyped quite heavily from different analysts and praised to be an architecture that is going to replace tightly coupled and heavy to maintain monolithically designed applications. Then the Event-Driven Architecture (EDA) was thrown into the pit [66]. The distinction and comparison of these two complementary concepts and their integration with each other is important to understand the basic differences, their special abilities, strengths and drawbacks.

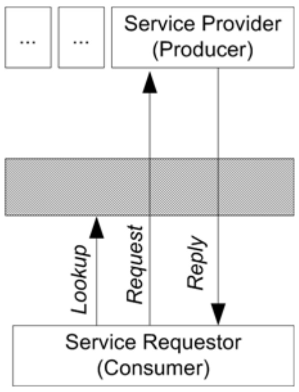


Figure 3.9: Service-Oriented Architecture Interaction Model

SOA is relying on the previously introduced request/reply mechanism where a consumer requests a service and awaits the reply from the service producer. The service-oriented architectural concepts are based on distributing and bundling services, in sense of making them accessible across organizational and technical boundaries. This type of architecture offers common interfaces, throughout heterogeneous platforms. The main goal is to accelerate the agility of software development and integration projects and thus enhancing the agility of organizations. SOA solutions are characterized as loosely coupled components, the communication is one-to-one and done in a synchronous way. Similar to RPC components in SOA-based solutions determine when another components service is called.

Figure 3.9 shows an illustration of a SOA based interaction model. A consumer or service-requestor is capable of looking up services that are registered and sharing a common interface towards service consumers. The consumer can call services through standard interfaces and and receives the results after the request has been processed and an answer has been generated by the service provider.

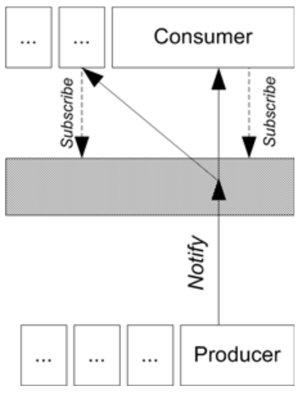


Figure 3.10: Event-Driven Architecture Interaction Model

3.1.7.2 Event-Driven Architecture

The event-driven architecture (EDA) is a complementary approach to SOA that can be described as an interaction model between decoupled and potentially heterogeneous components. The key in EDA is that components interact with each other through exchanging events that represent state changes in observations. The communication is done asynchronously, in contrast to SOA-based solutions, and therefore the communication parties don't have to be online and alive while exchanging information. The events are the communication drivers. If they occur all interested parties will be notified through many-to-many channels. A reliable, loosely coupled, scalable and fault tolerant middleware is required to support this type of architecture. The major difference to SOA is that the components are more loosely coupled and they are more fault tolerant through their asynchronous communication behavior. Mentioned earlier in this thesis, messaging middleware, in particular pub/sub messaging proofed to be one of the best solutions to support EDA.

Figure 3.10 shows an illustration of an EDA based interaction model. The consumers can register themselves through a mediator/middleware to events they want to receive notifications about. The producer publishes notifications of events while the mediator is responsible of re-distributing them to the interested parties. The figure shows an indication that EDA is strongly related to messaging middleware systems. Especially the publish/subscribe paradigm plays an important role with sophisticated subscription mechanisms.

3.1.7.3 SOA and EDA Comparison

The article [60] provides an interesting tabular comparison of SOA and EDA characteristics that pinpoints the main characteristics of SOA and EDA in the tables 3.1 and 3.2. The tables show the main highlights and differences of SOA and EDA at a glance. In the Table 3.1 however the attribute *Loosely coupled interactions* is not a dedicated SOA feature. Moreover EDA solutions are actually even more loosely coupled which is taken into account for EDA under the point *Decoupled interactions* in 3.2.

Gregor Hohpe compiled a list characteristic for EDA [50] that consists of the subsequent items. Most of the characteristics for EDA proposed by [60].

- **Broadcast Communication:** A communication model where a sender that is producing and sending out messages to multiple receivers. This point can be referred to the point many-to-many communication receivers.
- **Timeliness:** No intentional delays are involved. Events are fired whenever they occur. Hohpe is alluding to batch cycle processes with this point.
- **Asynchrony:** The producer and consumer don't have to be in sync. This point is mentioned in the Table 3.2 under the attribute Asynchronous.
- **Fine Grained Events:** Hohpe states that *applications tend to publish individual events as opposed to a single event* [50]. This might be true in a lot of cases, however especially in context of complex event processing (CEP) systems this item might not be fully accurate as it is capable of creating and maintaining relationships between events and aggregating them to higher level events. Under the last point in this list, Hohpe lists complex event processing as a key characteristic.

  CEP components can also take in the role of event producers. One of the abilities of CEP solutions is that they are capable of aggregating events to more higher and abstract levels. For instance the events *TransportStart* and *TransportEnd* could be aggregated to a higher level event *GoodsDeliveredInTime* which is fired if the agreed delivery time in *TransportStart* is smaller or equals the delivery time in *TransportEnd*. In this case the CEP component injects an event that has a lower granularity and aggregated attributes derived from lower-level events.
- **Ontology:** Ontologies or other an other categorization method that are used to classify events and determine the interest in group of events or events that fulfill some constraints.
- **Complex Event Processing:** This point mentions the ability to understand

and monitor relationships between events in terms of causality and creating aggregations [50][54].

Gregor Hohpe created an explanation of EDA in [50] that is using the programming metaphor call-stack to describe the features of EDA and creating a distinction to SOA. According to his point of view in [50], stack-based interactions consists of three main features: coordination, continuation and context. Call-stack based computing requires components to be in sync when executing or calling methods. This is directly resulting from tight coupling where one component must know details about the other component.

An additional layer can be put between the components to reduce the level of coupling, like a messaging solution that is an enabler for asynchronous communication. In this case the service requestor does not have to know which components provide the requested functionalities anymore. The downside is still that in SOA-based solutions the components still await a response on a request although they are more loosely coupled. If the request has been received the requestor can continue with the instructions on the "stack" for this fork. But requestor does not know which other component is satisfying its request.

EDA on the other hand is just producing events, not caring about which and how many components are consuming that information. Further it does not await any explicit response on a generated event notifications.

3.1.7.4 SOA and EDA Application Scenarios

The command and control or request/reply behavior of SOA suites best for integrating with existing functionalities by wrapping away or replacing legacy applications within organizations with a more flexible and sustainable solution. Application scenarios that require transactional capabilities in order to be able to roll back committed actions fit as well for SOA-based approaches.

EDA on the other hand comes into play with its strengths in long taking business processes that need to be offline from time to time due to their nature. For instance a workflow that is interrupted for input or other activities that will occur later on time need to be set to inactive. Therefore, the interacting systems collaborate with each other in an asynchronous way. If the state of the process or workflow interruption has changed the interacting components are informed through an event that is representing that state change. None of the components has to be aware of each other, moreover the event itself is driving the interaction. In contrast to EDA, the caller component must know the provider from which it requests a service.

Capability	**Description**
Loosely coupled interactions	Services are invoked independently of their technology and location
One-to-one communications	One specific service is invoked by one consumer at a time. The communications are bidirectional
Consumer-based trigger	The flow of control is initiated by the client (the service consumer)
Synchronous	Replies are sent back to the consumer in a synchronous way

Table 3.1: Fundamental SOA characteristics [60]

Capability	**Description**
Decoupled interactions	Event publishers are not aware of the existence of event subscribers
Many-to-many communications	Publish/Subscribe messaging where one specific event can impact many subscribers
Event-based trigger	Flow of control that is determined by the recipient, based on an event posted
Asynchronous	Supports asynchronous operations through event messaging

Table 3.2: Fundamental EDA characteristics [60]

EDA is especially helpful in application scenarios where different information sources must be attached for a *global* awareness [26] of ongoing state changes. This has to be done in order to be able to analyze ongoing situations that are reflected through events and to be able to react on them in a timely manner.

3.2 Event-Based Systems

Event-based systems are seeing increasingly widespread use in applications ranging from time-critical systems, system management and control, to e-commerce. Event-based systems can capture information from various sources (producers) and distribute it in a timely manner to interested consumers.

They can be used to integrate a wide range of components into a loosely-coupled distributed system with event producers which can be application components, post-commit triggers in a database, sensors, or system monitors, and event consumers which can be application components, device controllers, databases or workflow queues.

This section introduces generic concepts of event-based system to the reader. Further it describes the various participating components of such an event-based system and how they work together with the surrounding environment and infrastructure. SARI [81] is an event-based and complex event processing (CEP) solution that is used to describe and illustrate some of the concepts introduced in this section. SARI is capable of processing large amounts of events, providing facilities with the capability to monitor, steer and optimize business processes in real-time. It allows to observe relevant business events to identify exceptional situations, indicate opportunities or problems combined with low latency times in decision making for supportive or counter measures.

Event-based systems rely heavily on event-driven architectures, where each of the components interact with each other through exchanging events that represent state changes in observations. Events are the communication drivers in event-based systems and if they occur all interested parties will be notified about their creation. A key characteristic in event-based systems is that their communication model is asynchronous and therefore the communication parties don't have to be online and alive while exchanging information.

3.2.1 Events for Event-Based Systems versus Events in Software Engineering

Events are a common metaphor in graphical user interfaces (GUI) for representing user inputs (e.g. interactions with the graphical user interface - GUI). GUIs accept input from user controls and create, asynchronously notifications about action in the user interface that are referred to events.

Components of such GUI applications are registered to interested events and thus they are notified in case an event fires at the GUI. The main benefit of this approach is to decouple the components that interact with each other and thus to enhance the code quality in terms of maintainability and reusability. This type of events are called *local events*.

The Model-View-Controller (MVC) pattern is a common pattern in software engineering for assembling user interactions with the back-end. This pattern was originally popularized in Smalltalk and is now common practice. The MVC pattern consists of three components:

- **Model:** This component represents the domain knowledge that is reflected through a data structure. Often this component is used to encapsulate database access.
- **View:** This component is responsible of rendering the information and managing only the presentation of the user interface.
- **Controller:** The controller component is responsible of controlling the states and the workflow of the user inputs and the communication with backend components.

This small excursion to software-engineering patterns is necessary to clarify the domain of events and event-based systems that this thesis is addressing. In case of GUIs the MVC is handling the user interface through its controller components. The controller component is maintaining and managing the state of an application. If a state change triggers an event the corresponding controller component is notified. For example a user clicked a button on a web-page, which caused an event to fire that causes the execution of a database query. In this thesis such events are considered as local events.

The term *local event* is actually misleading as the event can lead to different and actually distributed actions. Therefore it is difficult to draw the line between between local events and events that are used in the domain of event-based systems. However, such user input events create one-to-one event notifications or potentially one-to-many if different components are updated. Either way the communication applied by this pattern is usually realized synchronously as the user awaits response on its request immediately.

When we talk about event-based systems throughout this thesis these local event handles are not in scope. This thesis is addressing event-based systems in a distributed environment, which allows multiple, distribute and possibly heterogeneous components to exchange notifications about events and provide an infrastructure to manage subscriptions. Distributed event-based systems can be seen as an extension to the local event notification model.

3.2.2 Components of an Event-Based System

The goal of this section is to give the reader an understanding what an event-based system is about, which components exist and how they work together. Further it aims at giving the reader the picture how all of the previously introduced topics fit together.

This section gives subsequently a detailed overview of an event-based system

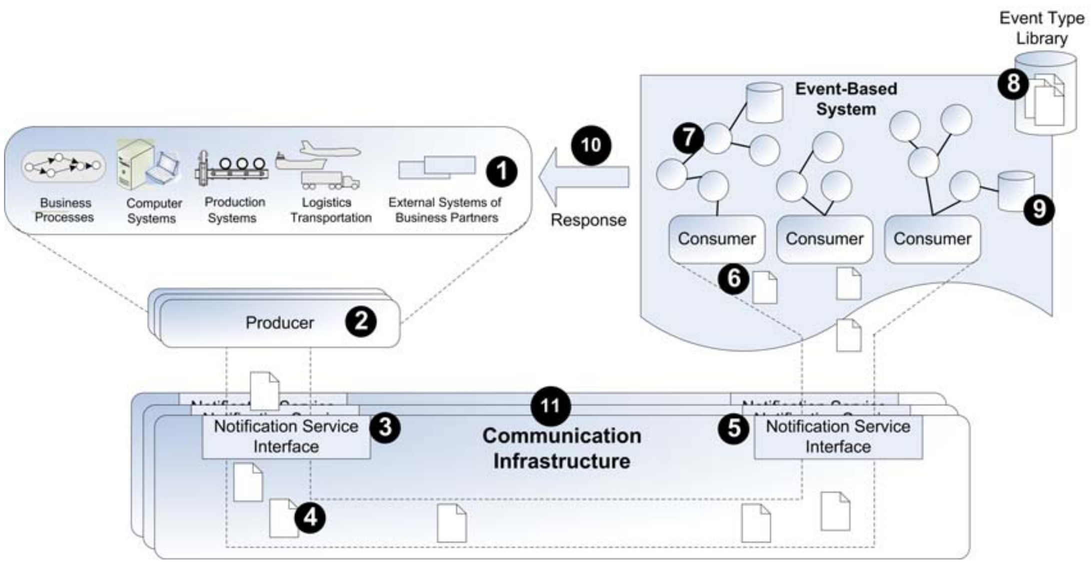


Figure 3.11: Event-Based Messaging Middleware

which is illustrated in Figure 3.11. The figure shows an event-based system consisting of several components that will be describe in details later on in this section. The message of this figure, and the introduction of its components, is to provide the reader with a delimitation of an event-based system in context of our research project SARI.

Event-based systems are characterized by having two interacting components like known from messaging solutions described in the previous sections - the consumer and the producer. Figure 3.11 illustrates the architecture of a distributed messaging solution that connects producers and consumers together through a communication infrastructure which can be seen as mediator between those two components.

The producer (2) takes relevant observations into account and decides if they should be published. The notification service interface, respectively the communication infrastructure, creates a notification including the event which represents the observation and distributes this event to the peers that might have an interest in this occurrence.

The peers can register their interest on events by subscribing to specific event notifications. The concrete registration and the expressiveness of formulating an interest by a consumer on a event notifications depends on the underlying communication infrastructure. In case of publish/subscribe solutions several methods have been described in Section 3.1.6.

The publish/subscribe paradigm has crystallized out to be a good choice for supporting event-driven solutions as they are loosely coupled, asynchronous and easily integratable into heterogeneous environments. Further they support different types of selecting and delivering event notifications to interested parties. For example a consumer can announce its interest in specific events or patterns of events (see also Section 3.1.6).

However, attaching an event-based system to several communication infrastructures requires own mechanisms to handle the subscription of events. Therefore, event-based systems (6) provide adapters which are consumers of event notifications and can correlate and aggregate events in order to discover and respond to event patterns. Event-based systems classify the events by using event types which are managed in an event type library (7).

The notification service interface (3)(5) is a prerequisite for loosely coupled systems since it provides a transparent abstraction layer for programming languages that hide the communication details. A channel (4) represents a bundle of event notifications that a consumer can subscribe to or a producer could select to distribute its notifications.

The following list contains an overview of all components of an event-based system. Each of the items will be described in details in the subsequent sections:

- **Event Sources (1):** The event sources represent the source systems and the outside world that is producing events.

- **Event Producers (2):** The event producers are the instances that publish event notifications about occurred events from the Event Sources.

- **Notification Service Interface (3)(5):** The notification service Interface is an interface towards the communication infrastructure and is used by the event producers to distribute their event notifications.

- **Event Notifications (4):** An event notification is generated by event producers and represents an event, defines the encoding and the data structure.

- **Communication infrastructure (11):** The communication infrastructure is the underlying communication infrastructure that is in charged of distributing Event notifications to the desired destination.

- **Event Consumers (6):** Event consumers are components of an event-based system. They can be seen adapters that are attached to the mediators whereas the notification services provide the interfaces towards the communication infrastructures.
- **Event Processing Realm (7):** The event processing realm of an event-based system represents the mechanisms and components for processing event notifications.
- **Event Type Library (8):** The event type library contains the information about the structure of event types for the event processing realm.

3.2.2.1 Event Sources

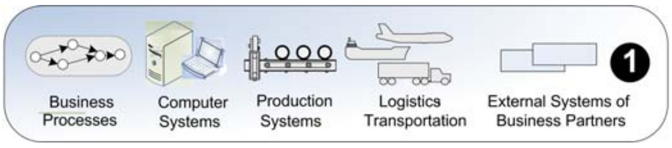


Figure 3.12: Typical Event Sources

Referring to Section 2.1 *Discussion About Events*, events can be defined *as observable actions or relevant state changes that can be absorbed by IT systems* in first place. Event sources can be arbitrarily everything that can be observed by a computer or an IT system. Such source systems can be for instance sensors, ERP systems, BPM systems, production systems or systems in charged of logistics operations. In Figure 3.11 event sources are represented in the upper left area (1) with symbolic representations of event sources like business processes, computer systems and so on.

An important point here is that the source system need some kind of a connection to the event producers (2), that take the observations into account and produce corresponding event notifications. The event sources and the producers are often tightly coupled as they need to be able to exchange the information that specifies an event. Basically event sources and event producers go hand in hand where the producer itself is the component that is actually creating the notification of an event whereas the event sources create the events itself.

3.2.2.2 Event Producers

Event producer components (see Figure 3.13 and Figure 3.11) take actions and relevant state changes from event sources into account and produce notifications of those events.

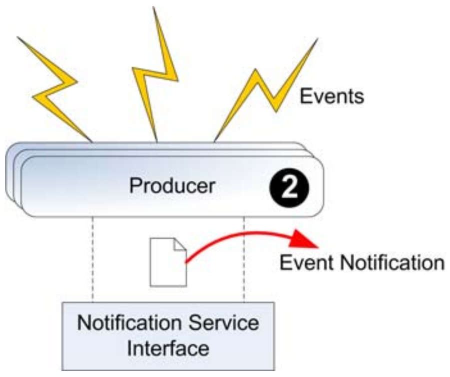


Figure 3.13: Event Producers

Event notifications are created by packing the observed events into messages according to a defined data structure which is also defined by the used interface of the underlying communication infrastructure. Such event notification messages contain a header and body with structured information that is often XML-based.

The event notification, created by an event producer component, is passed forward through an interface towards the underlying communication infrastructure that is taking care about handling the delivery of the message to its destination.

The event producer does not have any mediator capabilities, which is provided by middleware solutions for distributing those notifications and meeting the requirements of distributed systems. An event producer is only creating a notification and then using an interface for publishing or triggering the delivery of the event to interested parties. Failures from the producer components can not be captured by an event-based system at this place Therefore, the event-based systems processing realm must cope with possible malfunctions.

Going back to the discussion about event-driven architectures and service-oriented architectures in Section 3.1.7, one of the main characteristics to distinct the two architectural concepts is by the type of synchronicity and the degree of coupling between the communicating parties. Therefore, if the event producers work synchronously (e.g. sending off an event and waiting for a result to get back in sync with) they actually break with the concept of EDA. However, this is often the case in legacy applications and also with SOA-based applications which is not necessarily an issue for event-based systems in first place, but one for the architectural concept for whole solution landscapes.

In the event-based system component overview, presented in Figure 3.11, there is an arrow (10) directed from the event-based system towards the event sources (1). This arrow represents the characteristic that an event-based system itself can be both an event source and an event producer. The difference to other event sources and producers is, that the event-based systems event producers are within the scope of the event processing system itself. The event producer can be an integral part of the event processing system and attached to the event processing realm. This is because the event processing realm has its own event model, defining the syntax and semantics that are used to describe events, in order to be able to process them.

The result of the event processing in the processing realm can be another generated event that should be propagated back to other systems. To be able to emit created events it is necessary to transform the internal event data models into the required external data structure and then to pass it through to the implementation infrastructure (11). It is necessary to have mechanisms in place that allow to wrap the internal events into the correct format that is required by the targeted communication infrastructure. Depending on the targeted communication infrastructure for the event notification it is necessary to invoke the specific functionalities of that infrastructure which is exposed by the notification service interface.

In SARI, event producers within the scope of the event processing realm are called response adapters. The event adapter concept of SARI is discussed in detail in the Section 5.1. It basically consists of a transformation component that is in charge of converting and transforming the internal events to a desired target data structure and a response adapter that is then emitting the created event notification. This can be again another messaging middleware or in a simple case an insert of a record into a database.

3.2.2.3 Notification Service Interface

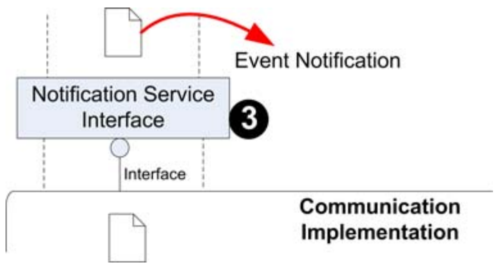


Figure 3.14: Notification Service Interface

The event notification service interface shown in Figure 3.14 is an interface between the event producers/consumers and the underlying communication infrastructure.

The event notification service interface is used by the communicating components (e.g. consumer and producer) to delivery and on the consumer side to receive the event notifications. In case of a consumer notification service interface, the expressiveness of advertising interests in events is limited by the communication infrastructure which is exposed by the interface.

Depending on the communication infrastructure that is attached to the event producers/consumers and the underlying technology (the programming language for instance), the notification service interface provides interfaces for the developers in order to enable the access to functionalities of the connected communication infrastructure.

In case of a publish/subscribe solution, the interface would provide access to functions like *adv(), pub(), sub(), unsub()* and *notify()*.

The reader should note at this point, that in this thesis the term event notification is used in terms of an interface towards an underlying communication infrastructure. In literature event notification components are sometimes associated as components containing already *intelligence* for distributing event notifications to the designated locations. Examples for notification services used in another meaning is the CORBA event notification [68], SIENA [23] which is an internet-scale system consisting of a multi-broker event notification service. Therefore in this thesis the term event notification interface is preferred instead of event notification.

3.2.2.4 Communication Infrastructure

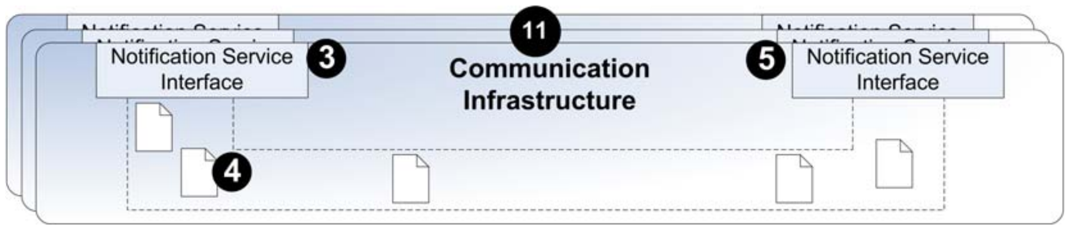

Notification Service Interface
3
11
Communication Infrastructure
5
Notification Service Interface
4

Figure 3.15: Communication Infrastructure components

The communication infrastructure (11) shown in Figure 3.15 is a component that is the core enabler for event-based systems. It provides the basic functionalities for exchanging events in a distributed and heterogeneous environment between event producers and consumers.

A communication infrastructure must satisfy the requirements for middleware systems to enable an easy integration of interacting components into a heterogeneous infrastructure. In this case the interacting components are the consumer (2) and producer (6) components which can be arbitrarily distributed and can be embedded into different types of environments. The communication infrastructure must fulfill the requirements based on [36] of network communication, coordination, reliability, scalability and heterogeneity introduced in Section 3.1.2.

In contrast to other work done in this field, especially work related to messaging middleware like pub/sub middleware solutions, tend to treat the communication infrastructure as a part of an event-based systems. The term communication infrastructure is often associated with the term *event service* [22][61] or *event notification service* [71][41].

So for instance [41] describes a notification service as a mediator which *alone is responsible for conveying notifications and it must deliver every published notification to all consumers having registered matching subscriptions* whereas the communication implementation itself is decoupled from the notification service. Pietzuch [71] however is not explicitly using the term of a notification service. In his work he is associating the communication infrastructure with the term *event brokers* that contain the complete implementation of the middleware parts.

People coming from the field of pub/sub middleware systems tend to consider the pub/sub as an essential part of an event-based system. Depending on the scoping of an event-based system this claim is not wrong. Even a primitive filter mechanism for event notifications can already be considered as a simple event processing system. Taking the more sophisticated approaches into account like type-based or content-based filter with an expressive subscription language we can already talk about event-based systems. Therefore, pub/sub middleware extended with sophisticated filtering/processing mechanisms are often considered to be an essential part of the event processing solution.

In Figure 3.15 the communication infrastructure is not labeled with an explicit communication paradigm and is set outside of the boundaries of the event-based systems. That is done because the communication paradigms and the infrastructure must be completely decoupled from the event processing realm. The connection between the event processing and the communication infrastructure is carried out through the event notification interfaces whereas SARI contains adapters (6) (called consumers in Figure 3.11) taking the role of event sinks to consume the incoming events.

Placing the communication infrastructure outside the boundaries of the event-based system, has the advantage that the event processing realm is not constrained by the limits of the underlying communication technology. Further it can be attached to other and different types of communication infrastructures. The major point here is that the event sinks/consumers of the event-based system can obtain the event notifications from arbitrarily different sources passed through different types of communication paradigms. This is especially useful in application scenarios with many different event sources in an heterogeneous IT environment.

If the communication infrastructure is an integral part of the event-based system it is more difficult to integrate a wide set of source systems. This can be a big issue especially with legacy systems. In case of a pub/sub middleware system, that has a content-based filtering extended with some other special capabilities like correlation, the event producer components of the event sources would have to be modified in order to integrate their event notifications into the event processing realm.

The drawback is with both approaches of scoping the components for event-based systems is that either way the event producers have to be able to interface with the communication middleware. In case of SARI it is easier to integrate event source as it already interface themselves with some kind of communication infrastructure as a lot of contemporary systems exchange information through standards. So there is often no big issue with attaching the event processing realm to the used communication infrastructure. If event sources don't explicitly use an appropriate communication infrastructure for event processing they might just store away events as records to a database. Then an event processing system, such as SARI, has to be attached directly to the corresponding database relations from where it can processes new records as events (a record represents one event). In that case the used communication infrastructure would be from a high level perspective the database system with its transactional capabilities.

For instance lets consider the scenario where there are not only messaging middleware systems, that attach the adapters of the event-based system to a queue. Lets imagine that the scenario requires to integrate sensors, like CCTV combined with motion detection. That system is potentially not open and does not have event producers out of the box that is implementing the notification service interfaces towards a messaging middleware. Potentially it is using a proprietary, encrypted and closed protocol to publish events that have been recognized by the motion detection software. Therefore it is necessary to be able to attach the event-based system to this type of event sources and infrastructure. In case the scenario requires more than one event source to be taken into account, it is easy to create new event sinks of the event-based system and attach them to the communication infrastructure used by the event sources. While on the other hand if the communication infrastructure, like pub/sub middleware, is an integral part of an event-based system, the event producers must be created or adapted that are capable to interface themselves (e.g. using the notification interface) with the underlying communication infrastructure. This can be a difficult task and can become a task with high efforts to perform modifications at the source-side.

3.2.2.5 Event Consumers

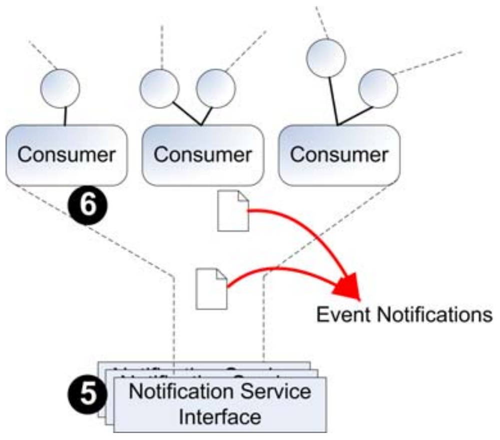


Figure 3.16: Event Consumer Components

The Figure 3.16 shows an illustration of the event consumer components (6) of an event-based system. The event consumer is the event sink of an event-based system and is connected to the event producers through a communication infrastructure (11) (see Figure 3.11) which is the mediator between those two interacting components. The mediator, described in the previous section, provides the functionalities and the infrastructure to deliver events from the event producer to the interested consumers.

A consumer can advertise its interest in specific event notifications by making use of the capabilities of the underlying infrastructure and in consequence the interfaces provided by the notification service interface (5). Depending on the communication infrastructure (to which the event consumer component is attached to) the notification service interface exposes the corresponding functionalities to the attached consumer. Therefore the expressiveness of creating subscription requests for a consumer depends on the mediator capabilities and its exposed functionalities.

In case of a pub/sub middleware system the consumer can place sophisticated subscriptions to receive event notifications. If another consumer however is registered to a simple POP3 mail service it can make use only of the limited functionalities expose by POP3. So for instance it can consume incoming e-mails, but filtering out messages for consumption with a specific sender name is not possible. This type of processing is then in scope of the event processing realm.

In this proposed event-based system component model the consumer is considered to be as an integral part of an event-based system. In SARI event consumers are called adapters. There are two types of adapters in SARI, there is a 1) sense adapter and a 2) response adapter. The first one is the equivalent to the event consumer in this component model and the second one is the event producer that is capable of propagating back events to the mediator. More on SARI's event processing architecture will be introduced and discussed in Section 5.1.

Further, in this proposed model an event notification is delivered through the mediator, the communication infrastructure, to the corresponding event sink. The exposed functionality by the notification service interface is used by the consumer to express its interest in specific events. The event notification delivered to the consumer consists of a specific data structure with a defined encoding. The data structure holds the actual content of an event and further it carries additional data for the middleware. This additional data is required by the middleware in order to be able to process and deliver the event notification. The consumers job in this event-based component model is to transform the encoded event notification into the event processing realm's own data structure. In case of SARI this is an event type library (8), shown in Figure 3.11. Based on the typing information of the event-based system, the consumer has to disassemble the event notification in order to transform the enclosed event data to a correct event type. SARI contains a concept of transformers that can be attached to the event adapters to perform this task. Details on this topic will be discussed in Section 5.1.

3.2.2.6 Event Type Library

The goal behind the concept of an event type library component (8), illustrated in Figure 3.17, is to provide a facility to define the schema and semantics for different types of events that should be valid in the given event processing realm. By creating and maintaining an event type library, the event processing realm can process events with data structures which correspond to uniquely addressable event types.

The main purpose of the event type library is to classify the event notifications, received by consumers, by using event types which are managed in an event type library.

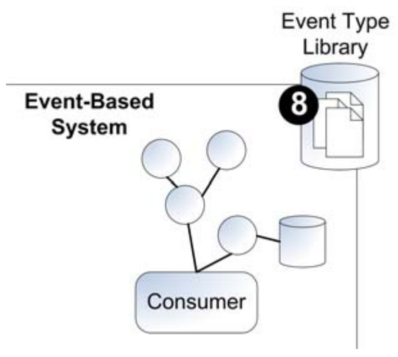


Figure 3.17: Event Type Library

Every event processing solution contains its own data structure to express the syntax and semantics of events that can be processed by the event processing engine. Some solutions have rather simple conventions and others have complex and rich underlying data models. Therefore, an event type library component can be found in many event processing solution, with respect to different kind of characteristics and thus it is a generic component of event-based systems.

A detailed discussion on event types in SARI can be found in Section 5.2.

3.2.2.7 Event Processing Realm

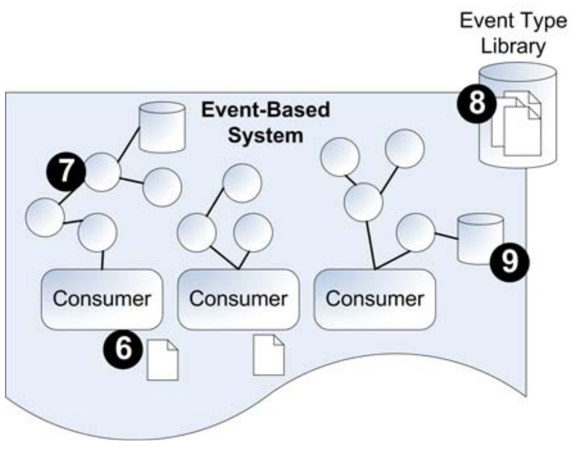


Figure 3.18: Event Processing Realm

Figure 3.18 shows a simplified illustration of an event-based system which is the core event processing component and defines the scoping of the interacting systems.

The scoping is defined in terms of which component can be seen as a part of the event-based system and which is outside the context of the event processing realm.

The event-based system, with its event processing realm, is docked to the "outside-world" through its event consumers and producers. The consumers (6) receives events through a communication infrastructure and transform them into their own event representation in terms of syntax and semantics. The data structures for events are maintained in the event type library (8), introduced in the previous Section (3.2.2.6), whereas the concrete manifestation of it depends on the event-based solution.

The communication from the event-based system to other components and the outside-world is conducted through event producer components. Referring to the sections about event consumer components 3.2.2.5 and event producing components 3.2.2.2 in SARI they are associated with the term adapters and sockets that will be discussed in detail in Section 5.1.

Basically every event processing system architecture contains some sort of an event sink that is receiving and consuming event notifications delivered through communication channels. In some solutions this consumer component might be strongly coupled with the event processing algorithm itself. For instance the consumer is responsible for receiving *OrderPlaced* events (e.g. representing a placed order for goods that should be delivered) and at the same time in charge of performing some action based on the attributes of the received event.

However, most of the solutions follow a flexible event processing approach which allow to model event processing steps or apply event rules. The event processing steps are represented by the connected circles (7) in Figure 3.18. SARI has event processing maps (described in details in Section 5.1.3) with various components, so called event services, that can perform different tasks on events. Further, there can be additional data sources (9) available to support the event processing procedures or to enrich events with additional information.

The conceptual representation of the event processing steps differs from solution to solution. However, most of these components can be found in some sort in all of them. The main goal is to give the reader of this thesis a basic sense for the big picture and potentials. In the subsequent chapters and sections several event-based systems and research projects will be introduced and described. Furthermore a special emphasis will be set at the research project SARI and the Event-Base concept.

3.3 Summary

In this chapter an event-based component model is introduced that set the communication infrastructure outside of the event-based systems boundaries. This decouples the event-based system completely from the communication infrastructure. Putting the communication infrastructure outside the boundaries of the event-based system, brings the advantage that the capabilities of the event processing realm are not constrained by the underlying communication technology. Furthermore, this component model allows to attach several and different types of communication infrastructures. This brings the advantage of consuming and processing events from a wide range of event sources.

Publish/subscribe middleware systems have revealed themselves to be a good choice to support event-driven architectures and event-based solutions as a communication middleware (e.g. mediator). Research coming from the field of pub/sub middleware systems tend to consider this communication paradigm as an essential part of an event-based system. Depending on the scoping of an event-based system it is not wrong. Even a primitive filter mechanism for event notifications can already be considered as a simple event processing system. Taking the more sophisticated approaches into account like type-based or content-based filter with an expressive subscription language we can already talk about event-based systems. Therefore pub/sub middleware, extended with sophisticated filtering/processing mechanisms, are often considered to be an essential part of the event processing solution.

4 Emerging Event Processing Paradigms

Introduced in Section 3.2, event-based systems gained an increasingly widespread use in a large set of applications, especially in time-critical systems, system management and control and e-commerce. Event-based systems can be attached to a number of heterogeneous and distributed source systems and dispatch events in a timely manner to interested consumers. They can be used to integrate a wide range of components into a loosely-coupled distributed system with event producers which can be application components, post-commit triggers in a database, sensors, or system monitors, and event consumers which can be application components, device controllers, databases, or workflow queues.

Nowadays, business processes evolved to networked workflows that are complex and executed in parallel with little human involvement to meet the needs of todayŠs agile and adaptive business environments [55]. The pillars of business models are loosely coupled, distributed and service-oriented or event driven systems that generate huge amounts of events at various granularity levels.

The lack of tracking these events and maintaining the causal relationships and traceability between those events, as well as aggregating them to high level events or correlating them, is a problem that is currently investigated by many research groups [6][54][79].

Contemporary business requirements yaw for agility, flexibility and service orientation. A simplified summarization of this wide discussed and necessary business trend can be reduced to the demand that today's businesses have to adapt their processes and organizations faster than their competitors. Business organizations that are capable to handle critical business events faster than their competitors will end up us winners in today's globalized market pace.

Traditional data warehousing approaches do a good job as a decision-making tool at a strategic level and for understanding the business situation based on a collection of historical data [47].

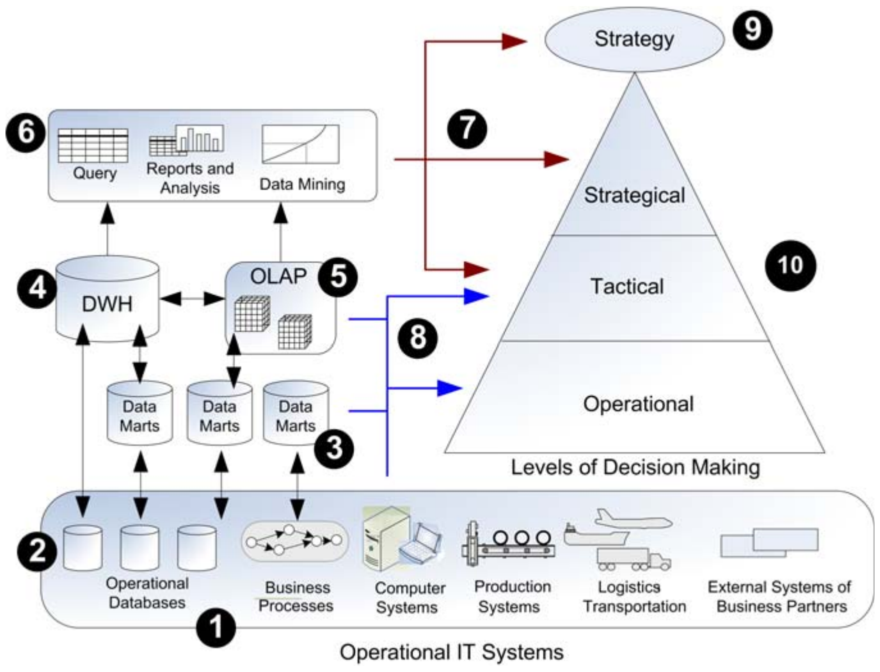


Figure 4.1: Decision Making Pyramid adapted from [47]

The IT systems of an enterprise define its ability to support closed loop decision making processes throughout its organizations. Figure 4.1 shows the loop of information that is currently in place in most of the organizations to support decision making. Figure 4.1 is an adapted illustration from Golfarelli et al [47]. At the bottom of the figure there is a list of exemplary IT systems from various organizations working together to accomplish a task. Further, most of these systems contain some sort of data storage managed by operational databases (2) (shown in the lower left side of the figure). All these systems can be attached to ETL (extract, transform and load) processes to extract data for either local data marts (3) of specific organizations or for data warehouses (4) containing global information. The data integration process (ETL) is a time and resource consuming task which is often performed regularly overnight in order to not disturb the daily business operations. Further, this task requires high efforts of adaption in case the underlying systems change their data structure.

To extract valuable information from the collected data it is required to perform analysis over the data collections. There is a wide range of business intelligence tools on the market to perform the tasks of data analysis (6). Users of such systems can place custom queries, create reports or dig around in the data by performing data mining tasks. By exercising these tasks the users of such systems are able to extract valuable information, perform analysis on them and at the end interpret the information and apply the gained insights to business.

Figure 4.1 shows a decision pyramid in relation with IT systems. This figure explains the flow and the sources of information that are required for the decision-making processes in an enterprise. On the right side there is a pyramid (10) representing the levels of decision making. On the very top, there is an enterprise vision and strategy (9) formulated by clear goals including objectives, concrete numbers and formulated means how to reach these goals - in short quantitative and qualitative goals underlined with actions to be taken to a certain level of abstraction.

As a necessary byproduct of a strategy there are means for measuring the performance and the status of the objectives of an organization. These key performance indicators (KPIs) are metrics to quantify the targets in order to be able to measure the progress and grasp the current status in reasonable information chunks. The KPIs are usually calculated out of data provided by the organizations IT systems like found in data warehouses. KPIs are one tool that can be used as an information source for making decisions. So for instance if the progress is not as expected in comparison to the KPI goals, decisions have to be made and actions need to be taken.

According to Golfarelli [47] the pyramid is described as follows:

> *At the strategic level, the global strategy of the enterprise is decided. The tactical level is usually composed by multiple divisions, each controlling a set of functions; the decisions taken here are related to the corresponding functions and must comply with the strategy defined at the upper level. Finally, at the operational level, the core activities are carried out; the decision power is limited to optimizing the specific production activities in accordance with the main strategy.*

Depending on the level of decision making information from different sources are required on different aggregation levels. The level of decision making defines also the requirements of the freshness of the data used for decision making. Decisions made on a strategic level require less fresh data than decisions made on an operation level. Further the time of making a decision is an important key factor on lower levels of the pyramid. Decisions that take long on operational levels bear the potential to cause severe losses of business opportunities. Of course this value depends on the concrete business case, but in general the manifestation of delayed decision has a higher negative impact on operational or tactical levels than on the strategic level.

Although analyses on historical data, using OLAP for instance, are a way to gain deeper insights, data warehouses are not meant to provide process knowledge in the first place and regularly they often donŠt provide data in a real-time fashion [76]. This leads to the lack of making decision on a tactical or operational level in a timely manner in order to preserve the value of the information until an action has been taken. Today business requires a fast paced decision making on operational and

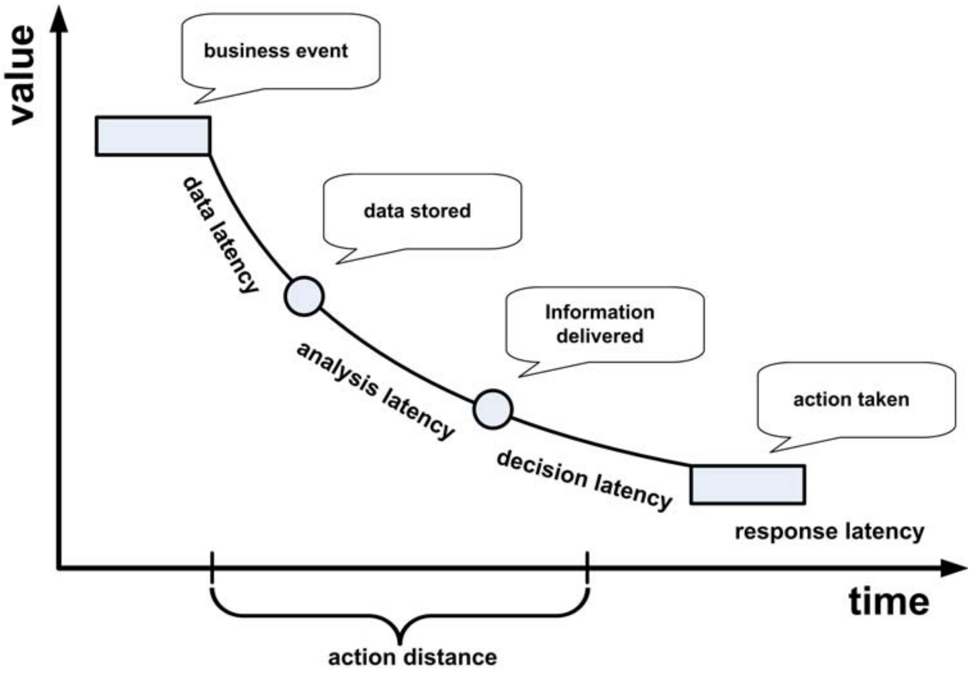


Figure 4.2: The decrease of the business value of events over time according to [49]

tactical decision making levels. Hackathorn presented in [49] an interesting graphic shown in Figure 4.2 that basically says that the business value of an event decreases by the amount of time that passes by until an appropriate action is taken. Of course in reality not every event has the same value loss over time, but basically the decrease of business value over time is a common characteristic.

The figure shows the four main latency drivers starting from the occurrence of the business event itself to the point when an action is taken.

- **Data Latency:** The data latency describes the time it takes to prepare the events for analysis. This is usually relating to the time it takes to store the events and to perform data transformation tasks.
- **Analysis Latency:** The analysis latency is the time that is required to complete

that analysis process to extract valuable and meaningful information out of the given data.

- **Decision Latency:** The decision latency is the time it takes for the decision makers to select the appropriate action that should be taken.
- **Response Latency:** The response latency covers the time for taking the decision and making the execution of the decided action possible.

One of the most promising concepts that approaches the problems of gaining real-time business knowledge, enable closed loop decision-making on operative levels and delivering real-time information on processes is complex event processing (CEP) and event stream processing (ESP).

The subsequent sections will try to clarify both terms that were hyped in certain industries in recent times. Because of the gaining popularity of those technologies both terms have been used by vendors in a confusing way. The marketeers of some commercial vendors that used to call their products event stream processing solution switch over to use the term of CEP now. Often the ESP is associated with high-volume and high-throughput attributes while CEP is often associate with more sophisticated event processing capabilities. Further the formal distinction of both techniques is still under heavy discussion in scientific communities.

In CEP/ESP-community discussions, CEP solutions are often described by their capability that allows the handling of *complex events* in terms of correlating them and creating abstractions (e.g. aggregations) while ESP solutions are considered to be linearly ordered. Linear ordering in context of ESP means that the incoming stream of events has some type of ordering which could be the time for instance. CEP solutions are considered to be a partially set of events sharing relationships like causality or other types.

Tim Bass introduced a good metaphor in a presentation at the DEBS07 conference [14] that explains the basic characteristics and differences between ESP and CEP.

In Figure 4.3 events are represented by dust particles flying around in the air. Those dust particles are the colored circles in the figure where each color represents a certain type of event. Now several air streams carry the particles into the cloud and form a partially ordered set of particles that are collected inside the cloud. In terms of ESP the streams of air that carry the particles into the cloud represent the event streams that ESP solutions are processing. The particles are ordered for instance by their arrival time in the cloud while the particles inside the cloud are randomly spread across the space at the first glance. The cloud is the space where CEP solutions

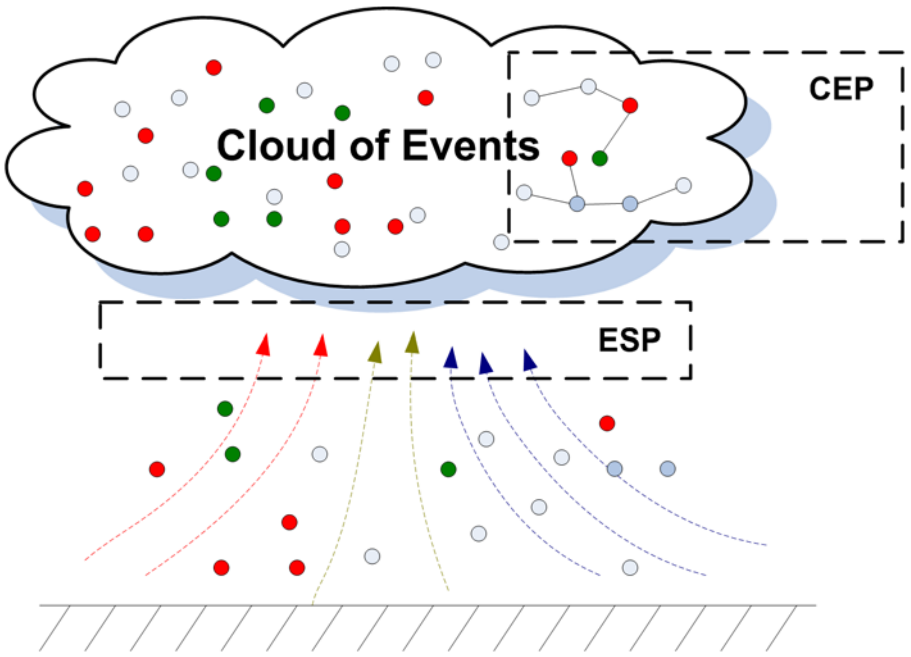


Figure 4.3: Metaphor to explain Complex Event Processing and Event Stream Processing. Illustration adapted from [14].

maintain the correlations or causal relationships between events, create aggregations and in general provide access to the *cloud of events*. A cloud may consist of the results from several streams (e.g. sources).

In the next two sections, a detailed explanation of CEP and ESP will be provided. As mentioned earlier these two terms are partly determined by marketeers nowadays. However, the sections will provide an explanation based on literature from the field and clarify the distinction of the two terms.

4.1 Complex Event Processing

The term of complex event processing (CEP) was first introduced by David Luckham in his book *The Power of Events* [54] and defines a set of technologies to process large amounts of events, utilizing them to monitor, steer and optimize the business in real time. The main application field for CEP is generally in areas where someone needs low latency times in decision cycles [49] combined with a high event throughput for observing relevant business events of predefined or exceptional situations, indicating opportunities or problems. Typically, these are areas like financial market analysis, trading, security, fraud detection, logistics like tracking shipments, compliance checks, customer care and relationship management and, in general, the monitoring of business processes and the reaction to them with a low delay.

A CEP system continuously processes and integrates the data included in events without any batch processes for extracting and loading data from different sources and storing it to a data warehouse for further processing or analysis. CEP solutions capture events from different sources, with different time order and take events with various relationships between each other into account.

In terms of business intelligence, active or zero latency data warehouses can significantly reduce the refresh cycles, but the missing process knowledge that is implicitly provided by events is still absent. The main gap that CEP solutions are addressing is the delay in analysis and thus, in taking actions, which can result in a loss of business value [49] (see Figure 4.2).

The key characteristic of a CEP system is its capability of handling complex event situations, detecting patterns, creating correlations, aggregating events and making use of time windows.

The question at this point is what is a complex event exactly and where is the line that separates a simple event from a complex event. The term of complex events was not mentioned in the previous Section 2.1.1 although it seems to be the core of complex event processing. Basically a complex event can be seen as an event that is an aggregation of other events whereas the CEP system is capable of handling, creating and processing such events.

According to the CEP glossary [56] Complex Event Processing is defined as:

> *Computing that performs operations on complex events, including reading, creating, transforming or abstracting them.*

The metaphorical Figure 4.3 shown in the introduction of this section is used to explain the difference between CEP and ESP. It shows a cloud where the dust particles in it represent the events and differentiate their types through their coloring. All of those events could come from different sources from IT systems in an enterprise. An arbitrarily different number of IT systems can be attached to event CEP Engine and thus filling the cloud with events at different granularity levels. That means that the cloud can contain low level events, like HTTP requests or already high level events that signalize an order made on the online web-shop. Depending on the event producer the events may differ in terms of they granularity and information density. Reduced to the essential task of a CEP solution, its job is to process those events, by creating and maintaining relationships between the events, aggregating them to meaning full levels, perform analyses, create enrichments, detect patterns and provide an infrastructure that allows to react on detected situations by propagating back actions to IT systems in the organizations.

Going back to the decision pyramid Figure 4.1 in Section 4 where the key message was that the tactical and operational decision making levels require a fast paced working with possibly fresh information. CEP solutions are capable of reducing the latency significantly if deployed properly.

CEP systems can handle the flood and the noise of events coming from operational systems and are able to bring in order, so that they are capable of making use of them. Based on this characteristic they have the capability to calculate for instance KPIs based on fresh data from operational systems and provide them directly to decision makers. Further, they are capable of enriching the information from historical sources (for instance gained insights from statistical analyses) and performing decisions and actions automatically if desired.

Looking at the pyramid Figure 4.1 a CEP solution is placed directly between the operational IT Systems (1), the data warehouses and the decision making instances. Further they are capable of covering the operational and tactical levels of decision making with various sophisticated techniques.

4.1.1 Example of a CEP Supported Scenario

For instance lets consider the given situation where different IT systems of a logistics company produce events relating to transports of goods. Figure 4.4 shows the source systems on the left side consisting of four divisions, with the assumption that their operation is supported by IT systems that are event producers. The logistics company has several warehouses spread across the country to be able to deliver goods in a fast and efficient manner to the customers (e.g. factories).

- **Warehouse:** Transports are loaded with the goods at the warehouse. If a shipment is created, loaded on a truck and ready for transport the event *ShipmentCreated* will be fired.

- **Truck:** The truck is the vehicle that is conducting the transport. It continuously sends its location (received through a GPS device) to the home base. If the truck is loaded and starts its route the *TransportStart* event is triggered. After the truck has unloaded the goods at the destination the *TransportEnd* event will be fired.

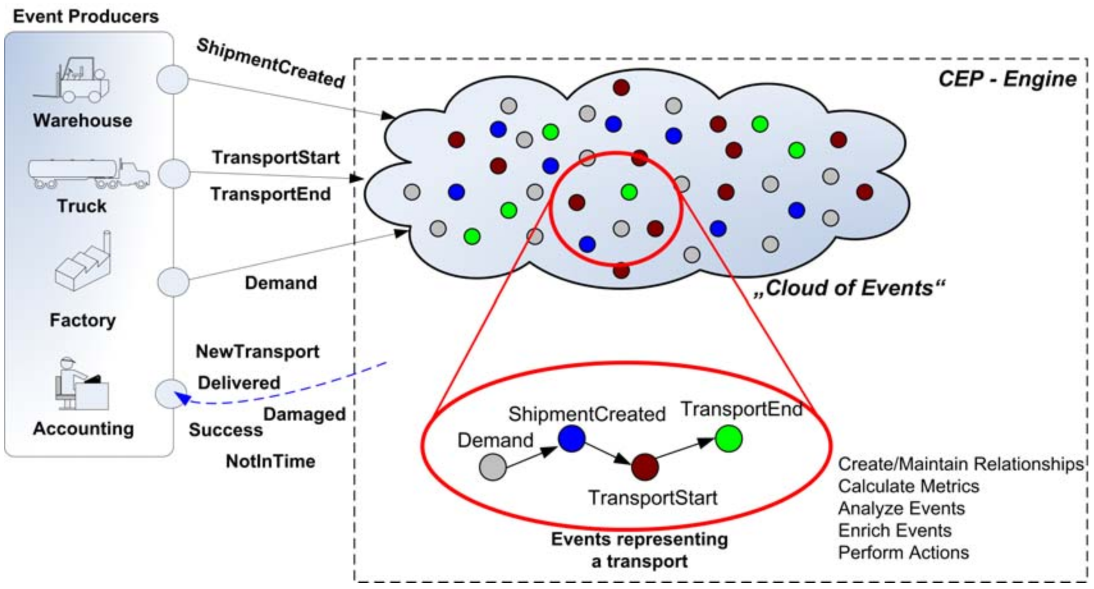


Figure 4.4: High Level Example of a Complex Event Processing Engine

- **Factory:** The factory's system is issuing Demand events for specific goods that should be delivered just-in-time to ensure the smoothness of the production. In case a demand is discovered the system triggers a Demand event containing the name of the goods, the amount and the due date that should be delivered to the location.
- **Accounting:** The accounting division, supported by an IT system, is receiving and evaluating the outcomes of transports. Following events are consumed and processed by the accounting system:
  - **NewTransport:** This event is created when a demand has been detected by the CEP-Engine containing the warehouse location from which the goods should be picked up and delivered to the which factory.
  - **Delivered:** The Delivered event is issued whenever a transport has reached its location and was unloaded from the truck.
  - **Damaged:** This event indicates the occurrence when the goods were unloaded, checked and they are damaged.
  - **Success:** This event indicates the occurrence when the goods were unloaded, checked and everything is fine.
  - **NotOnTime:** The *NotInTime* event indicates if the goods were delivered too late (e.g. after the due date/time of the delivery).

Whenever the CEP engine receives a Demand event for a specific good by a factory, it decides based on rules which location is the nearest one. It also checks if they have the goods in the right amount in place and triggers a *NewTransport* event. The *NewTransport* event is processed by the accounting and a new transport assignment is created. In case that the warehouses don't have a good capacity utilization the CEP system creates transports to relocate good surpluses from other warehouses in order to be able to satisfy the customer needs on time.

If the goods have been prepared for the transport and the truck starts the events *ShipmentCreated* and *TransportStarts* will be issued to the CEP engine. After the goods have been delivered and checked the CEP system sends events of the transport outcome. If a transport failed because the goods were damaged it can trigger a new transport to satisfy the customer needs. If the transport for a specific customer was several times in row not on time it triggers a promotion for that specific customer.

The point in this example is that it should show that various events are produced by different systems. All of them are attached to the CEP engine and form a kind of cloud. The CEP engine is capable of bringing an ordered into such huge noise of events. For instance a CEP engine should be able to maintain relations between events. In the simplest case it collects events relating to a transport like shown in Figure 4.4 where all the events belonging to a specific transport are highlighted. Based on those events it is capable of calculating metrics like the duration of that transport or the average transport duration of similar transports. Furthermore it is capable of analyzing those events and related events and is able to create aggregations out of them. For instance if a transport has been complete successfully it triggers an event *ShipmentCompleted* that contains the relevant aggregation of information from all the transport events.

4.2 Event Stream Processing

The roots of event stream processing (ESP) can be found in the field of relational databases and later in active- and zero latency data warehousing. Back in 1992 Terry et al [87] introduced their work on continuous queries. In their approach they provided the ability to specify so called continuous queries. Whenever a new record was added to the database it was analyzed and matched against a placed continuous query. If the query matched an interested user could be informed.

In traditional database approaches there are queries in place that can be executed on already persisted data records. In case of a usage scenario, where the records are updated with a high frequency and also inserted with high volume,

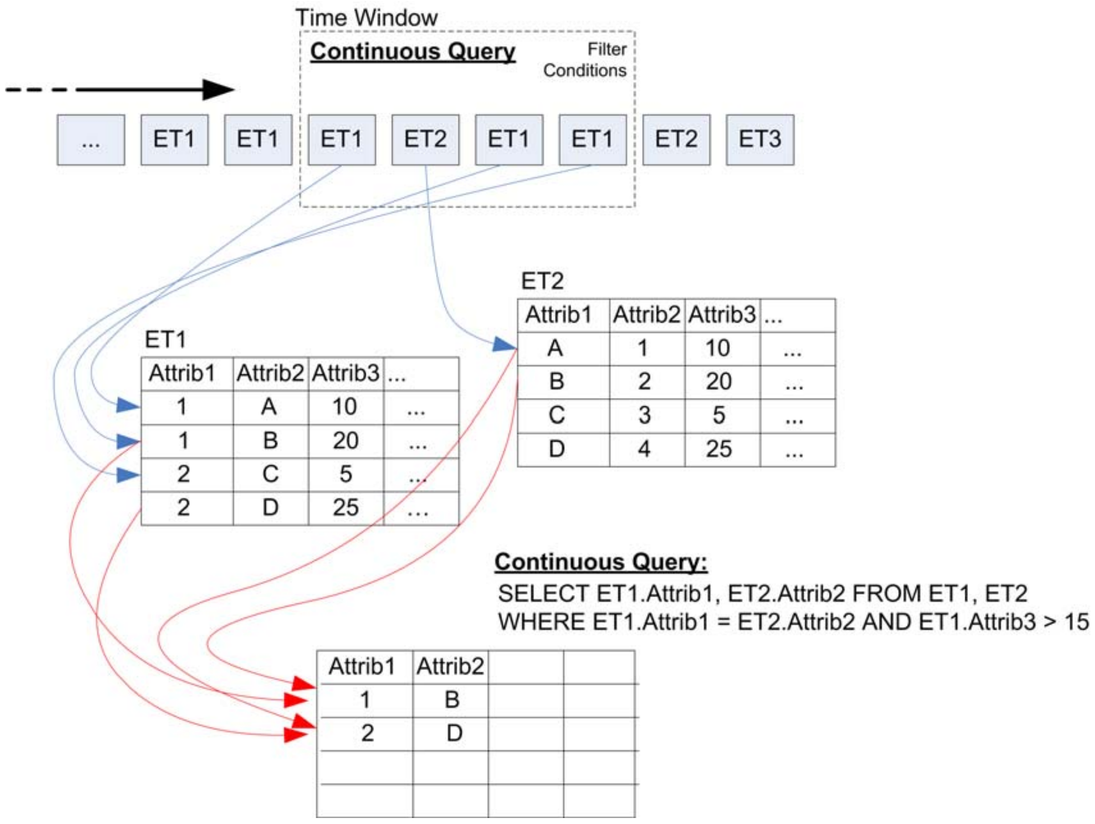


Figure 4.5: Event Stream Processing Example

it is hard to perform the queries to extract the desired information in a reasonable time.

The application scenario of this type of queries is in areas where data is changing constantly and comes in high volumes. This is the case especially in financial sector with stock tickers and sensor monitoring for instance.

Mentioned in the introduction of this section the terms event stream processing (ESP) and complex event processing (CEP) are a highly discussed topic in the event processing community. ESP solutions are characterized by processing a linearly ordered set of events. The linear ordering of events defines that a stream of events coming in into the ESP-Engine has an ordering by time for instance.

Shown in the metaphorical Figure 4.3 the streams are the red, green and blue dotted lines carrying up the events into a cloud of events. The ESP-Engine is set before the events are entering the cloud and plays out it's capabilities on the several streams of incoming events.

The CEP glossary [56] clarifies the ESP/CEP terminology with following comment:

> *The terminologies ESP and CEP are conceptual classifications. They can be useful in delineating philosophies of event processing and intended applications, but seldom specify accurately the underlying capabilities of event processing engines. It is possible for an event processing application to be reasonably described as both an ESP application and a CEP application.*[56]

Mark Palmer describes in an article [69] ESP as follows:

> *Event stream processing (ESP) deals with the task of processing multiple streams of event data with the goal of identifying the meaningful events within those streams with almost no latency - milliseconds matter for ESP.*[69]

Figure 4.5 shows an illustration of the event processing technique that most of the ESP-based solutions have in common. The first noticeable characteristic is that they are capable of attaching themselves, usually through adapters, to multiple streams of events. Each of the events flowing into the event processing realm has a corresponding type. In the example shown in the figure there are three event types labeled as *ET1*, *ET2* and *ET3* where each of them have several attributes representing the event data.

Continuous queries shown in this example can be defined on those incoming streams of events: In this query example the FROM clause contains the event types

```
SELECT ET1.Attrib1, ET2.Attrib2 FROM ET1, ET2
WHERE ET1.Attrib1 = ET2.Attrib2
AND ET1.Attrib3 > 15
```

Figure 4.6: Continuous Query Example

(here *ET1* and *ET2*) that should be taken out of the stream. The SELECT clause contains a projection known from ANSI SQL and a WHERE clause which sets filer constraints and a join between attributes of two event types.

For performance reasons ESP solutions often make use of a limited time frame. That means that the query is only applying the continuous query to a specified time interval. This could be for instance a query that is calculating the average amount of attributes in a time frame of 15 minutes. This type of frame limitation is necessary in order to keep the performance up, because otherwise the temporary operation tables will grow rapidly and especially join operations will take very long.

Let's consider the given example where the query picks out the event types *ET1* and *ET2*. It creates in-memory tables for those two event types and adds those records to that tables that are inside the specified time window. The attributes of the table are the event attributes. The query contains a join over *Attrib1* from *ET1* and *Attrib2* from *ET2* and creates a new table that will contain the query evaluation results according the defined projection.

This small example is just for illustration purposes for the reader of this thesis in order to give an impression how most of the ESP systems work. However not every ESP-based system is working internally the exact same way, but it pretty grasps the basic event processing concept. Further it explains why most of the ESP-based solutions provide SQL-like query languages. The reason is simply because they use as the underlying event model representation relational tables.

4.3 Summary

This section introduced the importance of IT systems that support closed loop decision making processes in order to satisfy the organizational requirements for agility, flexibility and service orientation. The level of decision making defines the requirements of the freshness of data. Traditional approaches such as data warehouses and OLAP do a good job to gain deeper insights based on a collection of historical data. However they don't provide data in a real-time fashion which is required on tactical and operational levels of decision making. This is an important issue as the value of a business event decreases by the amount of time that passes by until an appropriate action is taken. Therefore this chapter defined and discussed CEP, ESP and their differences. Both approaches offer the same event processing capabilities and are the most promising solutions for the class of problems of gaining real-time business knowledge, enable closed loop decision-making on operative levels and delivering real-time information on processes.

5 Event-Base

This chapter introduces the event-based system SARI and its extension the Event-Base. SARI [81] is a system that is capable of processing large amounts of events, providing facilities with the capability to monitor, steer and optimize business processes in real-time. It allows observing relevant business events to identify exceptional situations, indicate opportunities or problems combined with low latency times in decision making for supportive or counter measures. This chapter is intended to introduce the event processing models of SARI and their application. For a better understanding of the models and concepts are illustrated with a fictive fraud detection scenario.

The Event-Base extends SARI's event processing model with an efficient up-to-date operational storage together with retrieval mechanisms for business events for analytical as well as operational purposes. The query language for retrieving near real-time events and creating conjunctions with historical events, metrics and scores is SARI-SQL and is in contrast to Event Clouds indexing approach [76][73][90] a formally structured solution that extends ANSI-SQL. The language is tailored to satisfy the special requirements and meet the characteristics of events and their relationships.

This chapter also discusses special aspects of the event processing models in SARI and the Event-Base. The underlying event model and its various event typing concepts are discussed intensively as the design and the nature of the event model strongly constraints the capabilities of event processing query languages. Further, different event correlation concepts (e.g. mechanisms for creating and maintaining relationships between events) are introduced and described in detail. Special attention is set on event-driven rules, which are components that can be used to model decision trees with event actions in the event processing model. Looking deeper at this special concept, it reveals that they are using the domain-specific language EAExpressions for accessing and performing operations on events. That language is a subset of the SARI-SQL language introduced later on in this thesis. EAExpressions in event-driven rules allow to access event details and perform basic operations with a required Boolean return value. So for instance conditions, expressed with EAExpressions, can be applied on events with a resulting action. By extending this component with SARI-SQL it would be possible to additionally integrate historical event data, event correlations or metrics into the decision process. This integration with SARI-SQL significantly increases the expressiveness of event-driven rules and further they avoid the work around of creating enrichments steps of events before processing rules.

Finally, the data management concept of the Event-Base is described with a special focus on the internal data representation of events and their correlations. Topics are introduced how events and their correlations are processed and persisted to the underlying data structures and further how the events are maintained for access by SARI-SQL.

5.1 Architecture

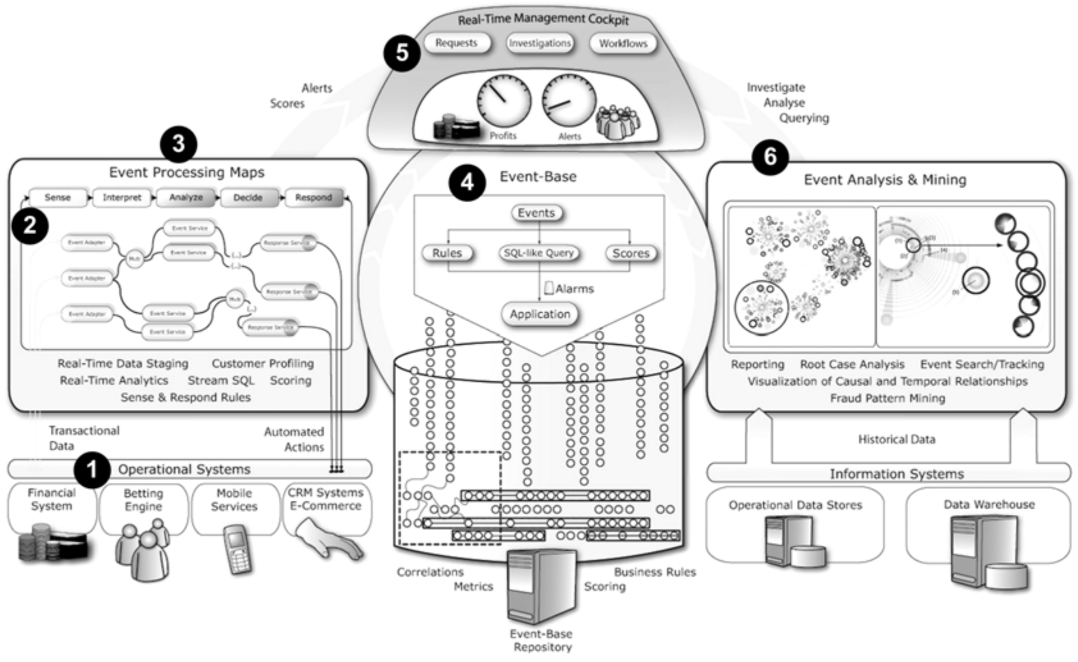


Figure 5.1: Event-Base high-level view architecture including its system components

The goal of this section is to give the reader an introduction of the architecture of the Event-Base starting with an overall high-level view of the system components and then dig down to details of the system architecture step-by-step in the subsequent sections. Each component of SARI, ranging from event adapters consuming events, to metric calculations and dashboard capabilities, are introduced and put into context of event-based systems discussed in section 3.2.2. Furthermore additional research efforts and components are introduced that are not an essential part of the Event-Base, but make use of its exposed capabilities and services. For instance analysis services can attach themselves to the Event-Base and query for interested data that is processed and persisted by the underlying systems.

In the following, the components will be reviewed subsequently and additional important concepts, like event models, of SARI will be discussed in detail. The concepts and models introduced are underlaid with the use case scenario of a the fraud detection solution [75] with the intention to provide an easy understandable and example-driven explanation. The Figure 5.1 shows the *big picture* of the Event-Base set into context of the fraud detection scenario. This fraud use case scenario follows an event-driven architecture (EDA) for preventing and managing fraud situations in the online betting domain. The architecture uses a rule-based solution for discovering and responding to fraud patterns. The system is supported by a business intelligence tool called EventTunnel for visualizing and analyzing fraud patterns, which allows users to investigate and discover fraud patterns. Thereby, the gained knowledge can be used to adapt or create rules to effectively close the gap between fraud analysis and fraud prevention.

In the following section the main building blocks belonging to the Event-Base will be discussed and introduced in detail:

- **Event Source/Operational Systems:** The operational systems create and emit events which are then absorbed by the event processing realm.
- **Event Adapters Sockets:** Events from the source systems are gathered through a communication infrastructure and transformed into a target typing schema.
- **Event Processing Maps:** The event processing maps are the main event processing instance and define the process of how events are evaluated.
- **Event-Base:** The event-base repository is the main storage for instance for maintaining and tracking event correlations, metrics and scores mainly for analysis and mining functionalities, but also for a continuous access and integration into the event processing maps.
- **Real-Time Management Cockpit:** The cockpit is a front-end component that can be used to present information about the current state of business.
- **Event Analysis and Mining:** Analysis and Mining tools are integrated with the Event-Base for discovering causal relationships between events, mining events, finding patterns within events and provide a facility for visualizing events, their relationships and patterns

5.1.1 Event Source/Operational Systems

The underlying business scenario, illustrated in the Figure 5.1, is based on an online gambling platform that consists of several system components. The group of the operational systems represent the event producers that can be attached to the event processing realm in either a push or pull way. Event producers, in context of event-based system component model, only create notifications and make use of interfaces to deliver the produced events. The communication mediator, which acts as a mediator between the event producing components and the event processing realm is not explicitly pictured in this illustration for purpose of simplicity.

However, in real-world application scenarios the various event producers might use different communication infrastructures to deliver their event notifications. Figure 5.1 shows the four event producing components of the gambling system on the bottom. A betting engine dealing with the processing of bets, a financial system processing financial transactions like cash-ins and cash-outs, mobile services for placing bets from mobile devices like cell phones and marketing relevant systems like CRM and other e-commerce related subsystems. So for instance, some of the event producer could use an enterprise service bus, while others make use of a messaging middleware.

Besides the operational event producing components, more static data sources could be integrated for purpose of enriching events with additional data during processing.

All of the event producing components are attached to SARI through adapters. SARI provides a flexible mechanism for docking the processing instances to source systems in a bi-directional way.

5.1.2 Event Adapter Sockets

The event producing components are interconnected with SARIs event processing engine through adapter sockets in a bi-directional way. Adapters can be docked directly to certain systems or communication infrastructures to sense (e.g. receive and consume) relevant events. These adapters are called sense adapters and are consumers of event notifications. The events are gathered either through a push or pull process that depends on the type of the system or the communication infrastructure. If the event producer for instance is an e-mail system, the adapters would pull out new mails in a periodically recurring cycle. The adapters contain transformations components that are in charge of converting and transforming the incoming event notifications to a given event object type.

On the other hand there are output adapters that re-inject specific events to the source systems to trigger some action in order to encounter detected situations like business opportunities. The output adapters are called respond adapters. The event-based architecture can be seamlessly integrated into distributed and service-oriented type of system landscapes. In case of legacy or proprietary systems SARI provides the facilities to easily add new adapters. The response adapters are also containing a transformation component to convert the internal events to a desired target data structure in ordered to emit the created event notification. This can be again another messaging middleware or in a simple case an insert of a record into a database.

5.1.3 Event Processing Maps

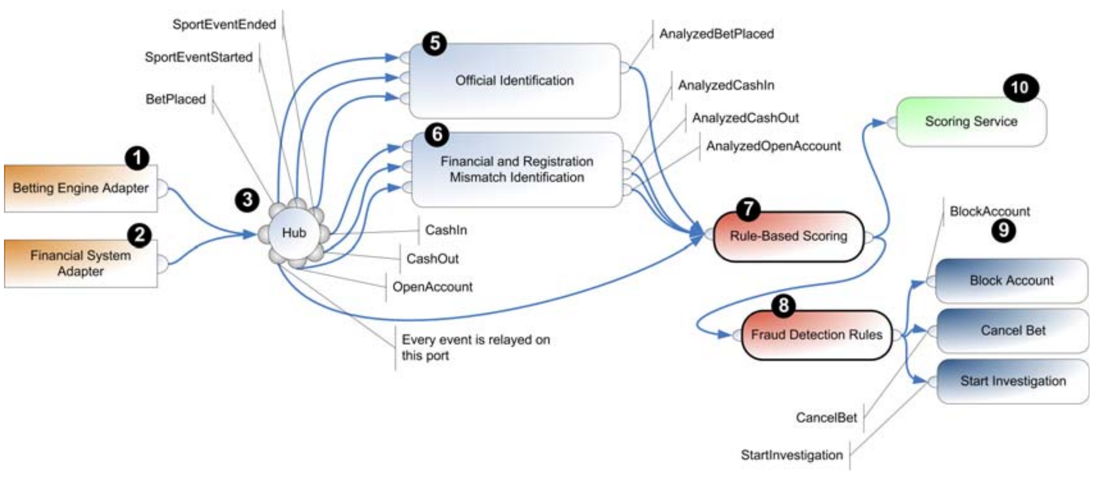


Figure 5.2: Event Processing Map

The actual event processing in SARI is executed within so called *Event Processing Maps (EPM)*. A typical map is shown in Figure 5.2 and consist of several different components that can be interwoven to define the process of how events are evaluated to satisfy the requirements of a scenario. Event processing maps and their components can be modeled and configured through a graphical user interface to create logical event processing flows to detect business scenarios, like opportunities or threats and trigger counter measures. An EPM allows to integrate multiple services and adapters which can be used to implement event-driven processes. Depending on the requirements and the business problem, the event services and adapters can be flexibly conjoined or disconnected. Links between the components and services represent a flow of events from one service to the next.

Figure 5.2 shows an event processing map for fraud detection use case with the intention to illustrate the concept of the EPMs. A processing map can be built by using adapters, event services and hubs. Each of those components can be interconnected to define and steer the flow of events.

5.1.3.1 Propagating Processing Maps with Events

Events are propagated through event adapters (1)(2) (see Figure 5.2) from various source systems into the event processing maps. Propagated events that are processed by adapters are transformed into event objects that match corresponding event type definitions [74].

In the fraud detection example, there are two event adapters where (1) is attached to the betting engine and (2) is attached to the financial system of the betting platform. The betting engine is in charge of handling all betting related issues like betting orders, processing bookmaking data and game tracking. The financial system is responsible of handling all kinds of money transaction which includes user cash-ins/cash outs that are especially relevant for fraud detection.

The adapters continuously receive events from the source systems, transform them to their according event types and propagate the instantiated event objects into the processing map. The source systems might produce a large number of events with different types, but only a part of these types are relevant for fraud detection processing.

5.1.3.2 Interconnecting Map Components

The round elements in the map in Figure 5.2 are hubs (3) that allow to route event streams from adapters to event services. Every event service must have one or more input and output ports for receiving and emitting events. Each port must correspond to an event type.

In this use case, there are dozens of events emitted by the event adapters from the source systems, but only several are required for detecting fraud in this processing map. The events *SportEventStarted, SportEventEnded, BetPlaced, CashIn, CashOut* and *OpenAccount* are routed to event services for special treatment.

The rounded rectangles in Figure 5.2 represent event services (5)(6)(9) that are custom components for processing events. This fraud detection use case has five different event services. The first two are used to analyse specific events and enrich them with data for further processing. The last three event services are responsive services that perform specific actions that are propagated back to source systems like the betting engine or the financial system.

5.1.3.3 Event Services

The event service *Official Identification* (5) can receive three types of events that are produced by the betting engine: *SportEventStarted, SportEventEnded* and *BetPlaced*. The *SportEvents* contain information about a started or finished sport event including a unique ID that references a sport event. The *BetPlaced* event contains information about the bet that has been placed by a user including information like an unique *BetID*, the users *AccountID*, a reference to the *SportEvent*, the amount and the odds for the bet that has been pre-calculated by the betting engine.

The *Official Identifications* task is to analyze those events for any betting activity of officials. There is a book making tool, at the backend of this service, that provides information about all officials including sport events with officials. When the user places a bet, the services checks if the user has an official role or function in that sport event where the bet was placed on. This kind of role or function could be for instance a player, a manager or a physiotherapist of a sport team. Such an official is allowed to place any bets at any time, but bets that are placed in a suspicious way, for instance on games, where he is involved, should be caught by the system. If the service catches an official that places a bet on an event, where he is involved, the *BetPlaced* event is enriched with the information that this bet has been identified as an official and emitted for further processing.

The event service *Financial and Registration Mismatch Identification* (6) job is to check financial related events (*CashIn, CashOut*) for any suspicious behavior. If a user creates transactions to or from accounts whose owner is an official, the service enriches the event with the information that an official has been detected; otherwise it adds information to the analyzed event that a payment name mismatch has been detected. This means if a user pays out money to an account whose owner differs to the registered user, the service automatically recognizes that there is a payment account/name mismatch. This is not fraudulent in first place but can be used to score the fraudulent behavior. On the other hand this event service checks every new registration (*OpenAccount*) for any official and matches this information with the payment information.

The event services *Block Account*, *Cancel Bet* and *Start Investigation* (9) receive events created by rule services previously in the map and perform actions. The *Block Account* event contains the *AccountID* of a user whose account should be blocked and the corresponding reason. The service propagates an action to the betting engine to block further actions by this user. The *StartInvestigation* event contains the *AccountID*, a reason and corresponding authority that should investigate the user. The message reveals detailed information about the reason of an investigation. Based on the previous rules the investigations are often combined with blocked accounts. The *CancelBet* event triggers the *CancelBet* service to perform an action that signalizes the betting engine to cancel the bet of a user.

5.1.3.4 Rule Services

The rounded rectangles (7)(8) in Figure 5.2 are rule services containing business rules to steer the process flow. The rules are comparable to decision trees to a limited extend and can be customized by the user through a graphical user interface. Like hubs, event services and rule services have ports that can be restricted to event types.

Behind the rule services, there is a sophisticated rule management system that is discussed in detail in Section 5.4 as it is an important concept for event processing and was introduced in the conference paper [80]. The rules are designed for business users and domain experts to allow them to model business situations.

The task of the *Rule-Based Scoring* service (7) is to check events for any anomalies and to create new scoring events that are routed to the Scoring Service (10). The rule service contains several rules that can be altered by business users or domain experts. For instance, there are rules that check if an *AnalyzedBetPlaced* event has been flagged as official. If such a rule matches a fraudulent behavior a domain user is able to weight the behavior within the rule.

The *Fraud Detection Rules* (8) rule service contains the rules for fraud detection based on tracked scores and other events that trigger conditions. Following up the example from the Rule-Based Scoring service where a detected official placed a bet and the users score has been increased significantly according to the rules specified. In this service if the same *PlacedBet* event is consumed a rule checks if the user with the *AccountID* from the *PlacedBet* has a high *FraudulentBehavior* score. In such a case, counter measures can be taken like canceling the bet, blocking the account and starting an investigation.

5.1.3.5 Scoring Services

The *Scoring Service* (10) in Figure 5.2 is a service that receives so-called scoring events. Basically, there are two main event types that are consumed by the Scoring Service. An *IncrementScore* lets the service increment the score value and a *DecrementScore* that decrements the specified score. A scoring event contains following attributes:

1. A *ScoreType* that is an identifier for the score.
2. A *Classifier* that can be used to relate a score to a user by its AccountID.
3. A *SubClassifier* that can be used to create sub category for a score.
4. A *Value* that increments or decrements the specified score corresponding to the classifier and sub classifier.

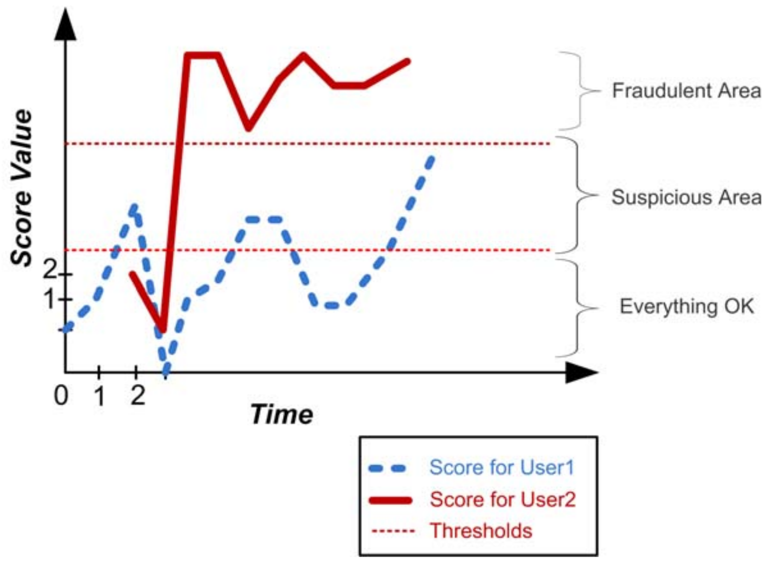


Figure 5.3: Scoring Example

Scoring can be used to provide an infrastructure for rating and tracking specific states throughout the event processing map. A concrete application scenario for scoring is the fraud use case, in which a *FraudulentBehavior* score is tracked for every user. The fraud potential of every user can be rated through domain experts by adjusting specific rules in rule service that can generate scores increasing or decreasing for users. If special events or event patterns occur and specific scores reached defined thresholds, it is possible to respond to the detected situations in real time.

For instance, if a user is regularly cashing in and out high amounts, the system increases the *FraudulentBehavior* score in small steps. This situation is basically nothing to worry about except the fact that the amounts are unusually high. Further, if the user always bets on games with extremely good odds, the score is increased further. If the user sticks to this behavior and the *FraudulentBehavior* score reaches a threshold, the system, depending on the rules, might inform an authority to start an investigation, because this might be a potential money laundering case.

5.1.4 Event-Base

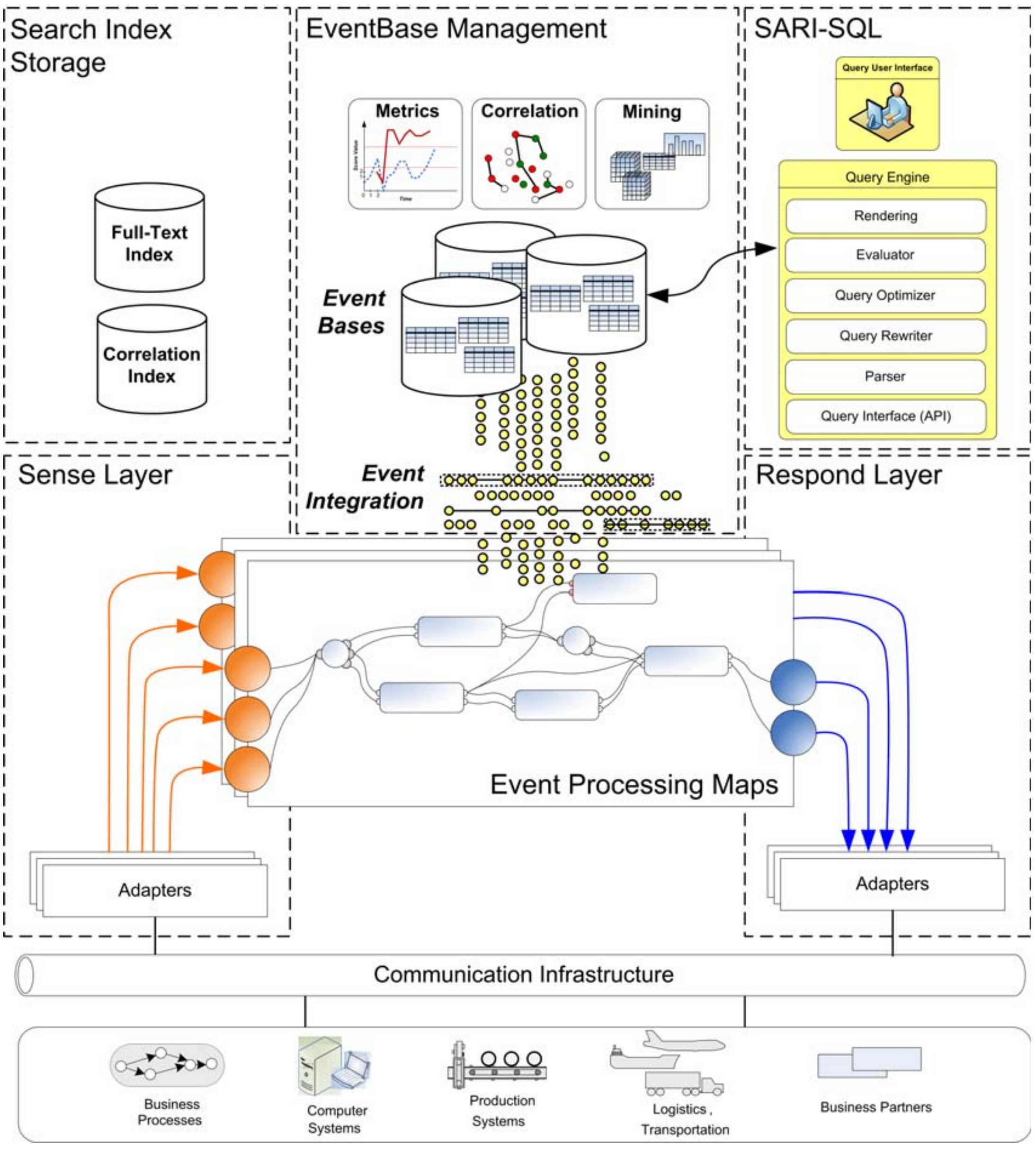


Figure 5.4: Event-Base Processing Schematics

The Event-Base is a data repository that can be considered as a next-generation database for the purpose of managing events, their correlations, continuously calculated metrics and correlations between events. It provides an efficient up-to-date operational storage together with retrieval mechanisms for business events for analytical as well as operational purposes without the costly data staging processes known from established data warehousing solutions. By providing access to such processed and prepared real-time events, it is possible to derive and generate new knowledge in order to provide facilities for decision making with low latency and thus allow to catch business opportunities or threats in a timely manner.

Current solutions for the analysis of business events are usually a combination of several tools, whereas the tools are not capable to provide a unified view on real-time events of an organization out of the box. This drawback of existing solutions means that the data integration of events requires large efforts and possibly extensive use of consulting services, and thus leads to high costs. Thus the adaptability of changing requirements is aggregated as well.

The Event-Base architecture, shown in a schematic illustration in Figure 5.4, provides, in contrast to other traditional data warehousing approaches, the ability to store and maintain events in real-time in a central repository, together with historical and analytical data. This type of persistent organization of events allows a seamless access towards real-time and historical event data. Furthermore the typical ETL steps of the data warehousing and analytical solutions are dispensed.

The Event-Base consists of following key characteristics:

Continuous data integration and adaption. Going back to the introduction of CEP/ESP in section 4 *Emerging Event Processing Paradigms* the Figure 4.1 of a Decision Making Pyramid shows the classical concept of an information system architecture. On the bottom of the picture, there are several operational systems whereas most of those system contain some sort of a data storage which is regularly an operational database. On top of those data storages and operational systems there are ETL processes to extract data for either local data marts (3) of specific organizations or for data warehouses.

In either case, such ETL processes are a highly time and resource consuming task and they are usually performed in cyclic recurring jobs in order to not disturb the daily business operations. These integration tasks are often done in nightly operations. Now in case of a change of the underlying systems or the data schema of the underlying data stores, adoptions in the above lying systems are difficult to make and require high efforts. The Event-Base is capable of adopting the event models automatically in case of a change or if new event types are added.

Query Language. The Event-Base provides a SQL-like query language for placing queries against near real-time events and historical events that have been processed by the Event-Base. Events captured from the event processing maps can be integrated into the Event-Base repository and accesses by the query language to retrieve correlated events, metrics and to perform event mining functions. In contrast to Event Cloud [76] the query language is formally structured to be able to exactly define a query with conditions to retrieve the desired events. Furthermore, SARI-SQL is decoupled from the internal data preparation and maintenance in contrast to the indexing approach of Event Cloud where the expressiveness of the search parameters is highly relying on the document based indexing approach [76][73][90].

Metric Calculation. The Event-Base allows its users to define metrics on specific events (e.g. event types) that are calculated dynamically during runtime. Such a metric for instance can be the average duration of transports. During a transport several events are involved and share a relationship (e.g. correlation over an *OrderID* or *ShipmentID* for instance). Therefore the Event-Base allows the definition of event correlations to express complex metrics. The execution of the calculation of metrics is managed by the system. Metrics can be calculated on 1) the occurrence of specific events, 2) by a user request, 3) periodically, 4) on request by a query or by 5) event data updates (e.g. data changes). In addition to metrics, that are defined by the users, the scoring is also an integral part of the Event-Base and can be seen as a derivative of metrics.

Event Mining Support. Event mining is an application of quantitative statistical methods on a set of events with the purpose of knowledge discovery. The result of an event-mining process is derived knowledge based on the interpretation of the mining activities and their results. This procedure requires event data to be selected and prepared. The Event-Base provides access to the event repositories to third party software components for such analysis purposes. The access itself is encapsulated through a data access layer that can be used to place SARI-SQL queries against processed events.

Response Mechanisms. Events of existing business environment can be received and consumed through event adapters that are then routed through the internal data exchange infrastructure for event integration and processing. The event processing and integration model is based on a sense and respond system. This allows to create new events in the processing model (for instance based on detected exceptional situations) and propagate them back to the business environments in order to trigger actions or countermeasures.

Distributed Data Management and Processing. The Event-Base is designed to conduct the event processing through a transparently distributed network of event processing and integration nodes. This ensures scalability, fault tolerance and enables load balancing. As a consequence it hides the complexity of distributed operations and thus allows developers of event-driven applications to focus directly on the problem domain.

The Figure 5.4 illustrates the overall components and the collaboration of the event processing instances within the Event-Base. On the bottom of the figure shown source system (i.e. the event producing components) continuously generate event notifications. The Sense Layer represents the adapters of SARI that can be docked to the event producing systems or the communication infrastructure. The adapters can gather events in either a push or pull process and propagate them into the event processing realms. The concrete event processing is performed in the event processing maps, previously introduced in 5.1.3, where the event processing flow is modeled with various components according to the business requirements.

The previous section 5.1.2 described SARIs adapter component model that are directly connected to the event processing maps. The Event-Base extends this concept with a logical layer of an *exchange infrastructure (XI)* separated into a sense and respond layer. This XI infrastructure can be seen as an event bus delivering the event notifications to the corresponding event consuming maps. The sockets allow to retrieve event notifications from adapters and redistribute them to arbitrary event processing maps. This allows a more flexible interconnection of event sources with the corresponding event consumers. For instance a socket that is connected to the event adapter of a source system A, retrieves all events from the source system. That particular socket can be taken and placed into event processing maps just like any event adapter. The difference is that the events are distributed to all sockets docked to the source system A through the XI infrastructure.

In the event processing engine, that evaluates the event processing maps, there is an *Event-Base Publishing Component* that is responsible for propagating the events into the Event-Base. This special event component can be arbitrarily connected, in a declarative way, to event services, hubs or adapters. An *Event-Base Publishing Component* is consuming events by applying filters, creating correlations between events, calculating metrics and then mapping those events with the preserved special event characteristics information to the Event-Bases data repository. This processes is highlighted in the Figure 5.4 by the term *Event Integration* which is represented as a flow of events from the event processing maps to the Event-Base repository for further application. During the flow of the events the correlations are created and the metrics are calculated.

The events branched off from the event processing maps are stored in a specific data structure introduced in the subsequent Section 5.5. Basically there can be created different numbers of Event-Bases for various event processing applications. Each of the Event-Bases contain application data and the infrastructure for processing events. The events stored to the Event-Base are preprocessed by a full-text indexing engine for providing a fast access to them and to preserve also the correlational relationships between the events. The processes of indexing events, their relationships and performance discussions can be found in [90] [76].

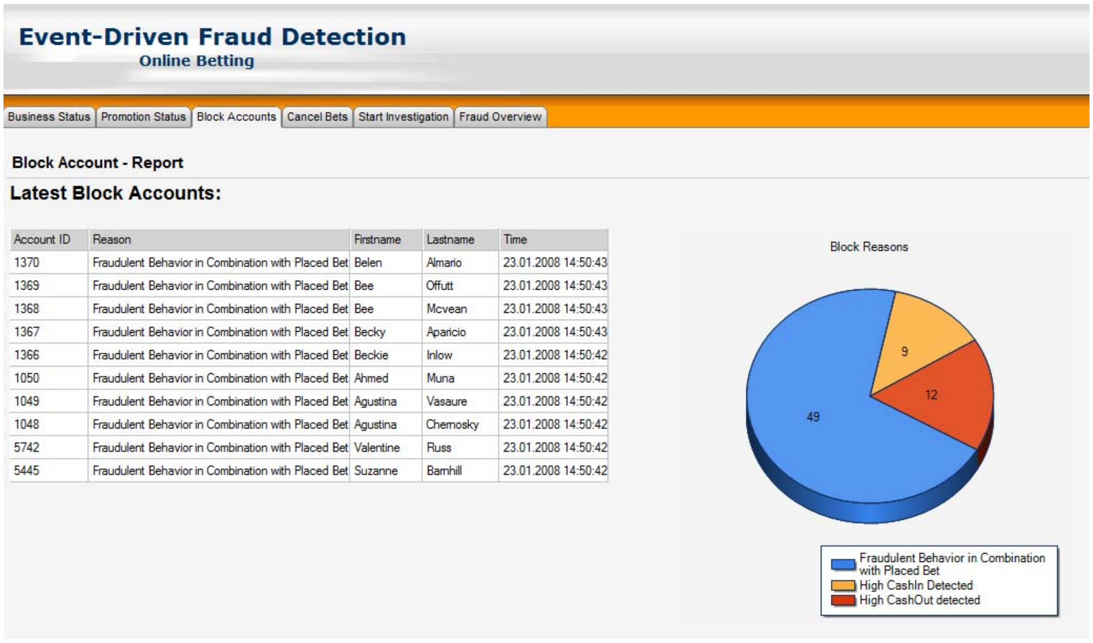
The core access component is a query engine, which supports SARI-SQL, that is set on top of the Event-Base repositories and exposes its services through programming interfaces and a graphical user interface. SARI-SQL is a structure query language and from syntactical point of view it is comparable to ANSI SQL. However SARI-SQL is extending the common SQL capabilities with special semantics that take the special nature of events into account. The query engine consists of six logical components, an API, a SARI-SQL language parser, a query-rewriter and -optimizer, a result evaluator and at the end a rendering engine for returning the results of a placed query. Details of SARI-SQL are under a detailed discussion in the section 6 Query Languages.

5.1.5 Real-Time Management Cockpit

The real-time management cockpit is a front-end component that can be used to present aggregated information about the current state of business. Dashboards or other information systems for data representation can access the Event-Base or display various kinds of information regarding the event processing stages. A good usage scenario are real-time dashboards for providing up-to-date information about the current fraudulent behaviors that have been detected by the system or to highlight sport events that might contain potential of abuse or service misusage.

The event processing map, shown in Figure 5.2, contains three response events (9) that trigger counter measures based on previously detected and evaluated fraudulent situations. So for instance an action can be triggered to block the account of a specific user that has shown a fraudulent behavior to a certain extent that exceeded the define score threshold limits. If such a “user block” action is executed the information about this situation, encapsulated in a *Block Account* event, is additionally stored as a record into a database. A dashboard application for monitoring fraudulent cases can now be built on top of such event processing cycles to accumulate, aggregate and present statistics or the states of the current business.

Figure 5.5 shows a screenshot of the management cockpit for the introduced fraud detection use-case. The left table displays the latest blocked accounts along with the reason. The right pie charts shows the distribution of the reasons of blocked accounts. This view is updated continuously while processing events in the event processing maps and the pane is updated in the second that an account is blocked.



Block Account - Report

Latest Block Accounts:

Account ID	Reason	Firstname	Lastname	Time
1370	Fraudulent Behavior in Combination with Placed Bet	Belen	Almario	23.01.2008 14:50:43
1369	Fraudulent Behavior in Combination with Placed Bet	Bee	Offutt	23.01.2008 14:50:43
1368	Fraudulent Behavior in Combination with Placed Bet	Bee	Mcvean	23.01.2008 14:50:43
1367	Fraudulent Behavior in Combination with Placed Bet	Becky	Aparicio	23.01.2008 14:50:43
1366	Fraudulent Behavior in Combination with Placed Bet	Beckie	Inlow	23.01.2008 14:50:42
1050	Fraudulent Behavior in Combination with Placed Bet	Ahmed	Muna	23.01.2008 14:50:42
1049	Fraudulent Behavior in Combination with Placed Bet	Agustina	Vasaure	23.01.2008 14:50:42
1048	Fraudulent Behavior in Combination with Placed Bet	Agustina	Chernosky	23.01.2008 14:50:42
5742	Fraudulent Behavior in Combination with Placed Bet	Valentine	Russ	23.01.2008 14:50:42
5445	Fraudulent Behavior in Combination with Placed Bet	Suzanne	Barnhill	23.01.2008 14:50:42

Figure 5.5: Cockpit Screenshot of the Fraud Use-Case

5.1.6 Event Analysis and Mining

Analysis and mining tools [85][76] are integrated with the Event-Base for discovering causal relationships between events, mining events, finding patterns within events and they provide a facility for visualizing events, their relationships and patterns.

Current solutions that exist today, especially in the field of business intelligence and data mining, relating to the analysis of historical data and/or the appliance of OLAP, are capable of providing deeper insights into business operations. However these traditional analysis tools are not capable of providing process knowledge and they regularly lack of providing data in a real time fashion. This leads also to a lack of decision making at operational and tactical levels in order to preserve the value of information which is decreasing by time [49]. Furthermore the traditional approaches don't consider the special characteristics of events that represent the flow of information in businesses, instead they align collections of historical data along dimensions for further evaluation. The relationships (e.g. correlations) and specific aspects of events are not available anymore in classical representations.

David Luckham presents in [54] a compilation of requirements for CEP analysis tools, whereas the requirements for those tools are addressed by the Event Analyzer. The list of requirements from Luckham were reduced by Vecera in [89] to the most relevant points.

- **Display parameters and attributes of an event.** This requires that a user needs to have facility to examine events, their attributes and their relationships at different granularity level without being overloaded. Further the item requires the (graphical) representation of events according their temporal order.

- **Trace the causal history of an event in an event execution.** Requirement that allows the backtracking of causal event chains that made a specific event to fire.

- **Graphically present events, event timelines and event correlation.** An event analysis tool should be capable of graphically represent events, their relationships to each other (e.g. correlations) and some way the temporal propagation during time.

- **Search Patterns.** This is one is possibly the mightiest functionalities that is required by an event analysis tool. An analysis tool should be capable of searching through large repositories of events in order to detect patterns that represent some sort of exceptional situations.

- **Drill-Down.** The tool should support event granularity abstraction mechanisms. Such mechanisms should allow the users to create mind-size large abstractions of given situations and then allow to drill down deeper to more detailed levels.

Event Cloud [76][73][90][89] was one of the first approaches to allow users to search for business events and patterns of business events within a repository of historical events. Event Cloud processes events, thereby creating an index for events and correlations between events in order to enable an effective event search. It provides a historic view of events with Şdrill-downŤ capabilities to explore and discover different aspects of business processes based on event correlations. Event Cloud allows users to investigate events, such as picking up single events and displaying their content and discovering related events or event patterns

The Event Analyzer is a research effort, coupled to SARI and the Event-Base, that provides a toolset, for visualizing streams of events making use of tunnel metaphor, to address following problem statements according Suntinger: *Where did irregularities occur in my business? Did processes change over time? Does my business* *slow down, or can certain processes be executed more effectively? Which of different execution paths of a process is most effective? Which contributors to my business process are most valuable? Upon which data were past, automated decisions made? Did errors occur in the automated decision process? What happened at a certain point in time at a certain location and who was involved?*[85].

The Event Analyzer is based on retrieving the desired information from the Event-Base through placing SARI-SQL queries against the Event-Base. By applying queries the Event Analyzer users are capable of retrieving events, persevered with their correlations, in order to create a tunnel like visualization.

The Figure 5.6 presented in [85] illustrates the basic concept of the visualization technique of the event tunnel. The top picture shows a cylinder where the events flow from the left to right in a temporal order. Correlations between events are represented through connectors between event instances. The top and the side views provide arrangement space for organizing events along the axes according to various placement policies to support pattern detection. A detailed discussion of event mining with the Event Analyzer can be found in [85].

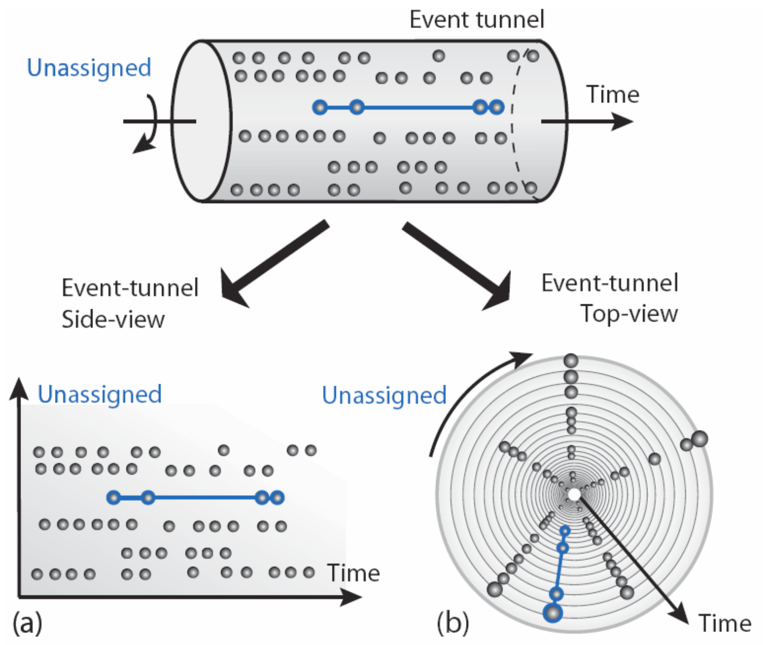


Figure 5.6: The event-tunnel visualization: Side view and top view onto the stream of events. The unassigned axis of a tunnel are utilized to show temporal relationships between correlated event sequences. [85]

5.2 Event Model Concepts

This section introduces the data structures that are representing the events and their attributes reflecting the content and the context of events. The event model of the event-based system SARI, is illustrated and its various event typing concepts are discussed in detail throughout this section.

The research on event-based systems focuses on typical usage patterns, such as the publish-subscribe paradigm as well as on stream processing, continuous queries, and rule-based event correlation. The developed systems used various ways of representing, filtering and querying events. In many cases, event models have been influenced by query languages, distributed platforms or architectures for integrating systems.

This section aims at describing and discussing the typing concepts and their applications within SARI. The design and the nature of the event model strongly influences the capabilities of event processing query languages [74]. Therefore this section pays attention to existing event models in the field, in order to highlight the model capabilities of SARI in contrast to other approaches and solutions. This is especially important to understand the key features and benefits of the event access language EAExpressions and the query language SARI-SQL that will be introduced in the next chapters.

The key characteristics of CEP/ESP systems are their capabilities of handling complex event situations, detecting patterns, creating correlations, aggregating events and making use of time windows.

Research has shown that the underlying event model can provide insights into the capabilities of event processing engines. Despite the discussions about a distinction between CEP and ESP solutions, the event models have shown that ESP solutions usually treat events in the form of tuples, while CEP solutions make use of more complex data structures.

Compared to other IT systems, event-based systems still lack in the support of tools that allow users to easily reconfigure a system or to refactor service and components. An event model has a major impact on the flexibility and usability of tools. In the following, the role and importance of an event model will be shown with some illustrative examples.

- **Development Tools:** An event model can be used by development tools to check the consistency of linked processing tasks or for offering auto-completetition capabilities (which are common in integrated development environments) for event-related expressions. Such capabilities can significantly facilitate the definition queries, event-triggered rules or data mappings.

- **Integration Tools:** Typically, the producer of an event is a business system which has to be integrated with the event-based system. With the rising popularity of XML, events are often represented as XML messages which are sent from a producer to a consumer. Many existing event-based systems have originally started with a SQL-based approach for querying event streams which come along with certain limitations when querying hierarchical data.

- **Event Data Management:** The integration and management of events, that paves the way for event analysis applications, is strongly influenced by the capabilities of the event typing model.

- **Event Mining:** An event model has a significant impact on how event patterns can be discovered in event streams or within historical event traces [65][59]. For many statistical analyses, it is necessary to capture sample sets for events that have specific characteristics. Thereby, many event mining approaches require event types for classifying, ranking or analyzing temporal sequence patterns.

- **Query and Rule Management:** The definition of queries and rules is in many event-based systems challenging for users. Graphical tools for building queries and rules require an event model that is easy to understand for (business) users.

Decoupling producers and consumers leads to a number of advantages previously discussed in Section 3 Event-Based Messaging Middleware. However, as they share a common interest in exchanging data about notified occurrences the producer does not deal with any details of consumers for processing the events. The drawback is that the consumer must find a way to understand received events what entails the need for an universal event model.

An event model tries to address this problem by providing a definition of event types with a detailed description of the nature and structure for the events. On the one hand, event types facilitate the event processing on the consumer side, but on the other hand, it raises the question how to deal with unknown attributes in events. Enabling unknown attributes or untyped events in the event model requires a strategy to deal with potentially missing event attributes and ambiguous event types.

The scope of an event model is defined as follows:

- Description of a valid schema that defines the attributes of events and can be identified by an event processing consumer.

- Mechanisms to extend an existing schema for aggregating or specializing events.
- Possibility to classify event types that can be processed by the event consumer for subscription purposes.
- Definition of relationships between events.

Historically, the first event models arose in business activity monitoring and complex event processing. Early models haven been mainly influenced by publish/subscribe systems, in particular those which supported content-based filtering mechanisms [51].

Distributed pub/sub architectures such as Hermes [8], Gryphon [51][11], and Siena [23] only provide parameterized primitive events. Since these systems focus on implementing messaging middleware which delivers events to consumers based upon their previously specified interest, however leave the event processing to the application programmer. Hermes [8] is a distributed event-based middleware architecture making use of a typed event model and supporting features known from object-oriented programming languages.

The type-based publish/subscribe was first introduced by Eugster [39][38]. Eugster proposes an event type model that integrates with the type model of an object-oriented programming language. Events are treated as first-class (Java) objects, and subscribers specify the class of objects they are willing to receive. No attribute-based filtering is supported, as this would break encapsulation principles. Instead, arbitrary methods can be called on the event object to provide a filtering condition.

Chen et al. [27] show an approach for rule-based event correlation. In their approach, they correlate and adapt complex/structural XML events corresponding to an XML schema. The authors describe an approach for translating hierarchical structured events into an event model which uses name-value pairs for storing event attributes.

Esper [30] is an Open Source event stream processing solution for analyzing event streams. The event processing engine represents its events through simple POJOs whereas the attributes and their types are determined through the Java class attributes. It supports also nested type structures through dictionaries or collections and supports inheritance structures that are considered by the query statements. Esper furthermore requires an event type definition in XML and thus supports validation through XML. It is possible to access events through the event query language, through XPath or programmatically through Java code.

AMIT [3] is an event stream engine whose goal is to provide high-performance situation detection mechanisms. AMIT distincts between “concrete events” and “inferred events”. The “concrete events” represents state change that has been observed in the real-world while the “inferred event” is an artificially created event such as a conclusion drawn out of real-world events. An event is the base entity and can be specified with a set of typed attributes. Events are represented through event instances which can share relationships among each other. Every event corresponds to an event type that defines the schema of attributes.

Borealis and Aurora [2] are stream processing engines for SQL-based queries over streaming data with efficient scheduling and QoS delivery mechanisms. Medusa [92] focuses on extending Auroras stream processing engine to distribute the event processing. Auroras stream data model is represented by a sequence of tuples, containing attribute-value pairs, where each tuple is marked by a timestamps that defines the entrance time of the event in the aurora network. Borealis extends Aurora’s event stream processing engines with dynamic query modification and revision capabilities and makes use of Medusas distributed extensions. Borealis data model was extend in contrast to Aurora with the ability to support the revision of data from incoming streams. Therefore there are now three types of message available. An insertion message (which are tuples known from Aurora), Deletion messages and replacement messages. The processing graphs from Aurora have been reused and are explained in Section 7.6.

The next sections will introduce various event model concepts and illustrate them with examples from the SARI system [81].

5.2.1 Event Type Model

The centerpiece of the event model in SARI is a library which manages the event types of a system. The library is a repository which manages metadata for describing events and their attributes. An event definition that is maintained in the library is further called an *event object type* in SARI. The concrete instantiation of events during runtime are *event objects* that must be valid to a defined event object type.

The idea behind the library is to provide a facility to specify the schema and semantics for all types of events that should be valid in the given event processing system. By using event type libraries, it is possible to create uniquely addressable event types that can be used to match incoming events to its own processing realm in an event-based application that is deployed in the system. The library stores event object types in a database and allows to share them among a set of applications or systems.

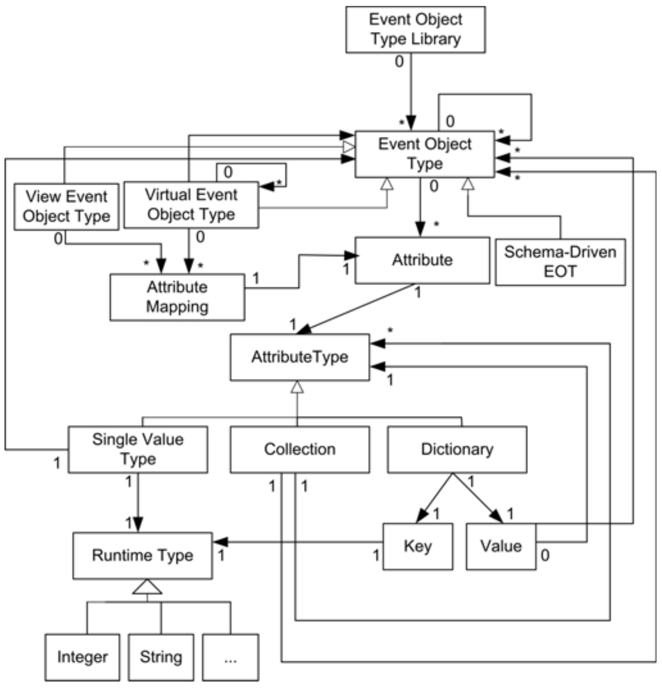


Figure 5.7: Event Object Type Model

Figure 5.7 shows the meta model of SARI's event object type library and event object type meta model. An event object type library can contain a set of event object types. An event object type can be inherited from other event object types. Event object types can contain several attributes. Each of those attributes have to correspond to an attribute type. An attribute type can be either a single-value type, collection type or a dictionary. A special case of an event object type is a schema-driven EOT. This event object type has all characteristic of its super-type. The difference is that these types and their attributes are generated out of a schema definition. This is a useful feature that allows to exchange typing definitions between systems.

At the lowest level, an attribute value has to correspond to a runtime type (Integer, String, ...). A more advanced concept that can be realized within SARI is exheritance where virtual or view event objects can be created with attributes that are mapped to attributes of existing event object types. Exheritance and view/virtual event object types will be explained in detail in the subsequent sections. The definition of an event object type contains following items:

- event object type namespace

- event object type name and display name
- event implementation type
- attributes of the event object type
- parent event object type (in case of an inheritance)
- event object types which are virtual roots (in case of an exheritance)

The event object type namespace and the event object name make the event object type globally unique. This allows the reuse of event object types across multiple event-based systems. SARI's event model supports type inheritances for specializing event object types by inheriting its attributes. Every event object type and attribute type is identified by a uniform resource identifier (URI), which allows to define namespaces for event types in a way similar to programming languages (e.g. Java namespaces).

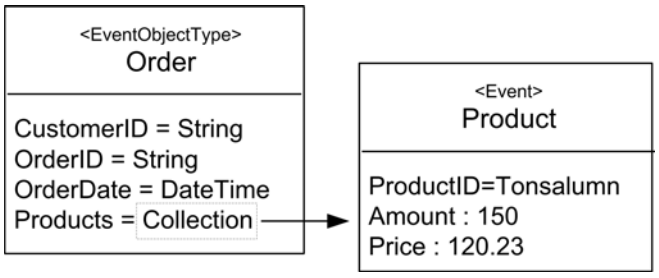


Figure 5.8: Event Object Type Example

Figure 5.8 shows a definition of an order event object type. The order event object type contains four attributes. *CustomerID*, *OrderID* and *OrderDate* have a runtime type. In addition, the attribute Products has a collection type, whose items must correspond to the Product event object type.

Figure 5.9 shows the previously defined event object type during runtime. The event is instantiated and filled with data according to its definition. Further, the collection of the attribute Products contains a set of products which are instances of the event object type Product. SARI allows to define event object types which can be used to define hierarchically structured events.

5.2.2 Attribute Model

SARI's event model supports the following three event attribute types: *single-value types*, *collection types* and *dictionary types*. The subsequent section will illustrate the

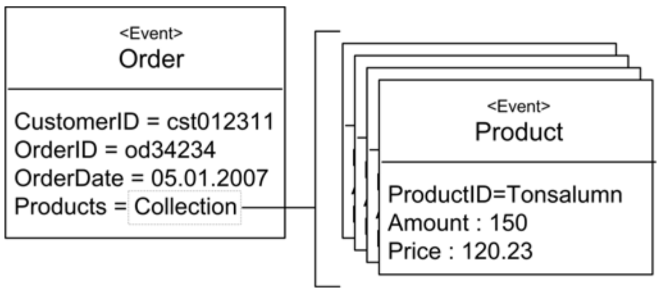


Figure 5.9: Event Object at Runtime

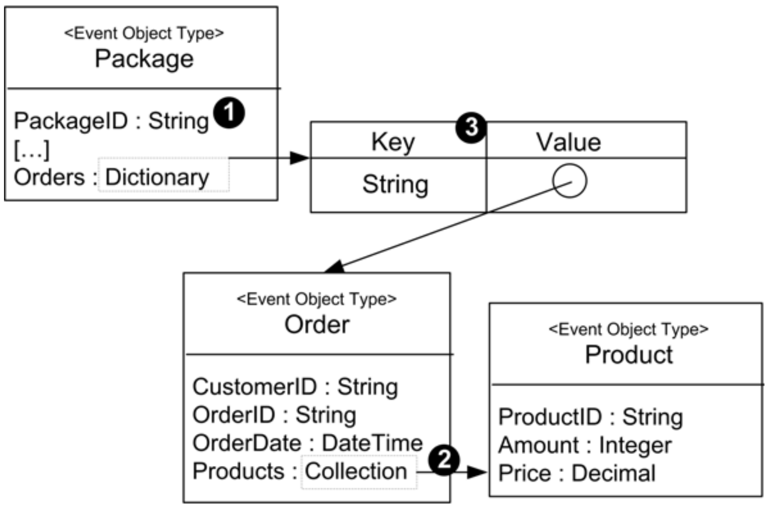


Figure 5.10: Attribute Model Example

supported event attribute types through an example presented in Figure 5.10.

- **Single-Value Types:** Single-value types represent attributes which have a run-time type such as a string, character, numeric and Boolean values. Additionally, single-value types can also represent another event object type (e.g. a customer event object type within an *Order Submitted* event object type). Figure 5.10 (Point 1) shows the attribute *PackageID* which is a single-value type of the run-time String type.

- **Collection Types:** Collections contain lists of values either corresponding to runtime types or to event object types. In either way, the collection values have to be typed. Figure 5.10 (Point 2) shows a collection attribute *Products*, where the items of the collection are typed as single-value types of the event object type Product. This attribute is a collection that can hold a list of *Product* event objects during runtime.

- **Dictionary Types:** Dictionaries are lists of key-value pairs containing either a value represented as a runtime type or as an event object type. The key, which is the accessor for the list element, must be defined as a runtime type. Figure 5.10 (Point 3) represents a dictionary attribute type containing a String as the key type and the Product event object type as value types. Please note that it is possible to nest every available type into dictionaries and collections as value types. By defining a collection type instead as the value element, the key-value pairs would contain a substructure holding collections.

5.2.3 Inheritance

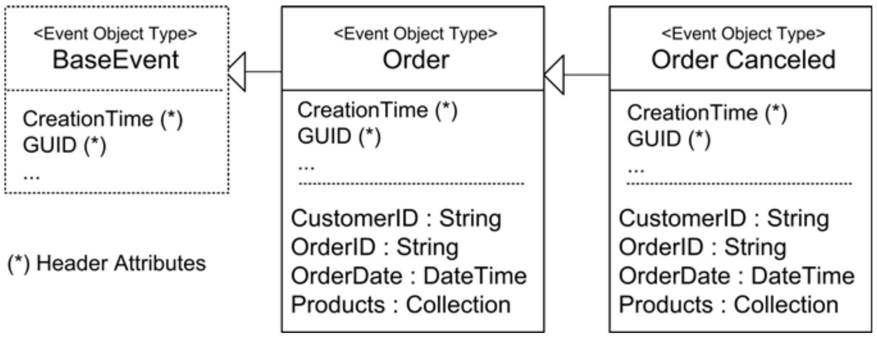


Figure 5.11: Inheritance Example

The root of all event object types is the *BaseEvent* type from where every event object type is derived. The *BaseEvent* type defines the header attributes that every event object type must have. SARI's event model allows specializing event object types by inheriting attributes from parent types similar to object-oriented programming languages. When using inheritance, it is not allowed to define a new attribute which has the same name as an attribute in a parent type. In SARI, inheritance can be defined by simply defining a parent URI in the event object type definition.

Figure 5.11 shows a simple inheritance example where an *OrderCanceled* event object type is inherited from an *Order* event object type. Both event object types are inherited from the *BaseEvent* type.

5.2.4 Exheritance

Another supported concept is exheritance which allows to create generalizations from event object types (see also [77] and [70]). Exheritance is the opposite of inheritance and allows to define generalization of event object types without modifying any existing event object type. In SARI, exheritance is provided by *virtual event object types*

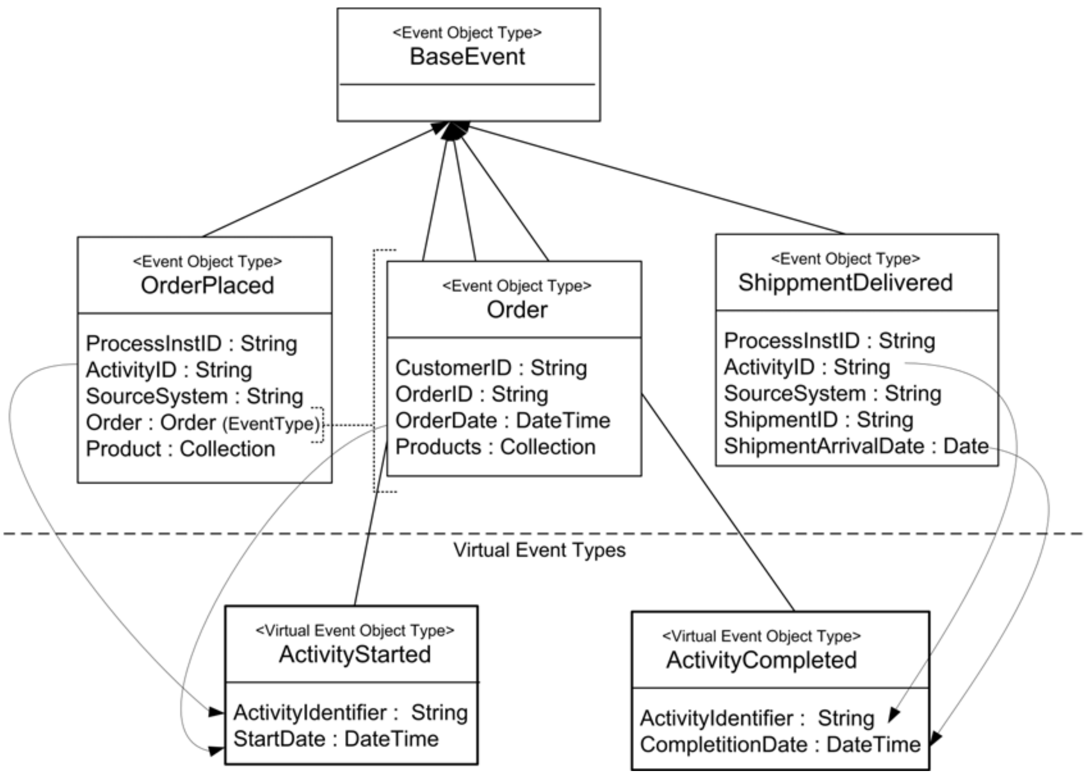


Figure 5.12: Exheritance Example

and *view event object types* which can be used to generalize existing types by mapping their attributes. The attributes of virtual event object types can be seen as a view on attributes of the generalized event object types that are mapped explicitly to a new type. Figure 5.12 shows an example containing three different event object types (*OrderPlaced, ShipmentShipped* and *ShipmentDelivered*). By making use of virtual event object types it is possible to generalize several attributes to a new event object type. In this example, the attributes from *OrderPlaced* and *Order* are generalized to a new *ActivityStarted* event object type and *ShipmentDelivered* form a new *ActivityCompleted* event object type. The view event object calculates the mapping automatically by inferencing the attributes with the same name and type of event object types.

5.2.5 Duck Typing

The concept of duck typing goes back to the support of dynamic typing in programming language concepts. Duck typing allows to implement interfaces dynamically at runtime and is part of programming languages like Ruby and Python.

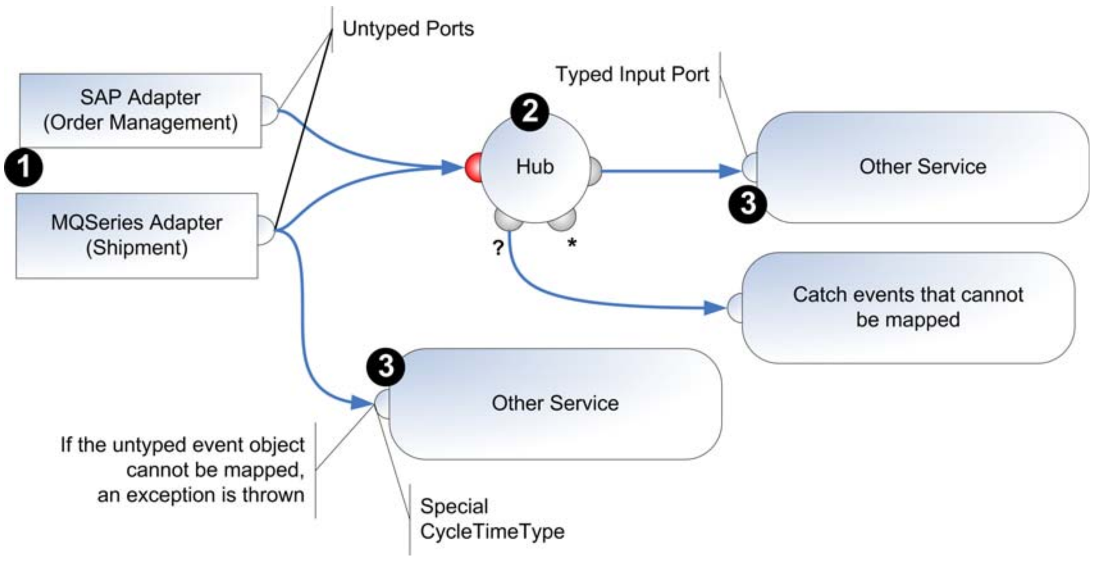

Untyped Ports
Typed Input Port
SAP Adapter (Order Management)
MQSeries Adapter (Shipment)
1
2
Hub
3
Other Service
?
*
Catch events that cannot be mapped
3
Other Service
If the untyped event object cannot be mapped, an exception is thrown
Special CycleTimeType

Figure 5.13: DuckTyping in Event Processing Maps

If it walks like a duck and quacks like a duck, it must be a duck. [44]

The idea behind duck typing for events is to allow event services to process untyped event objects. By untyped events it is meant that the event object type has not been previously defined in the event object type library. Every event in SARI is automatically derived from the *BaseEvent* type without declaration. The *BaseEvent* type enforces only header attributes for the event object during runtime.

The advantage of applying duck typing to event processing systems is to allow the developers of event-driven applications to directly reuse event data from source systems for processing. Otherwise the raw event data would have to be mapped to the event typing realm of the event processing system. This allows developers to focus on the problem domain in first place and allows to make changes during runtime without investing high efforts in integration tasks.

Figure 5.13 shows an example of duck typing in SARI. SARI uses event processing maps (see Figure 5.13) for processing event objects with event services (represented as rounded rectangles in Figure 5.13). Hubs (2) allow to route event streams from adapters and event services. Every event service must have one or more input and output ports for receiving and emitting events. Each port must correspond to an event object. By enabling duck typing of events, it is possible to emit un-typed events to typed ports. The system automatically checks whether un-typed event object are compatible to the event object type of a service port. Hubs can be used as routing mechanism to find out whether the system has not been able to infer a type from an un-typed event object.

Figure 5.13 illustrates this mechanism by propagating events from two different sources: an *Order Management* source and from a *Shipment* source (1). The input sources emit un-typed event objects. Point 3 in Figure 5.13 shows the input ports for event services whose ports expect typed event objects as input. By inferencing the incoming event object type, the service is able to process previously un-typed events during runtime. Hubs can be used to capture and reroute events which do not correspond to the input port of the event service.

5.2.6 Extensibility

Event object types are instantiated during runtime as event objects according to their type definition. SARI's event object types can allow "unknown" attributes, which allows to add any attribute with arbitrary attribute name and value to the event object. This is especially helpful if an event processing application wants to enrich the events with information additional information that is not known in advance.

As mentioned before, every event object in SARI contains event header attributes which are inherited from the BaseEvent type. Any event object type can add additional header attributes for capturing metadata about the event object.

5.2.7 Namespacing and Addressing

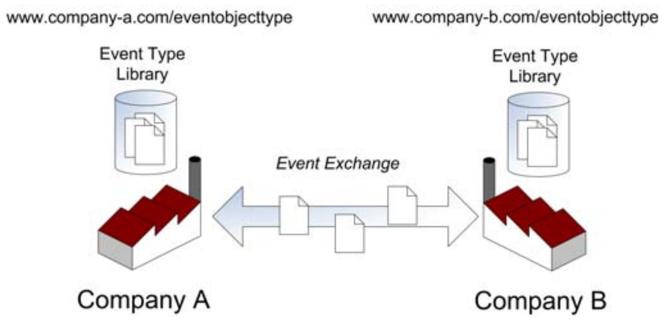


Figure 5.14: Sharing Event Object Types

Namespaces have found their way into object-oriented programming languages to organize components and its resources. SARI uses URI namespace for every event object type and for every attribute type. The objective behind using URIs as event object type and attribute identifier is to provide an unambiguous way to globally address types and attributes. Globally unique identifiers allow reusing event object types across system borders.

Figure 5.14 shows an example of two companies exchanging events. Each of those companies has their own event-based system with event object type libraries. Namespaces not only help to structure applications inside the company, they also help to share event object types between multiple organizations. In this example, the companies can share the event object types of both libraries. In such a business scenario, it is crucial that event object types are globally unique.

5.2.8 Language for Accessing Event Objects

When processing events, various ways for accessing the data in event objects are required. There is no single language which can sufficiently cover all requirements for processing events (such as listed above). Therefore, SARI supports the following 3 ways for accessing event objects:

- Event-Access Expressions (EA-Expressions) which have been designed for business users in order to flexibly access event data and to perform calculations. For instance, SARI uses EAExpressions for defining rules and for mapping event data to database tables, as well as for defining metrics and filters for analysis purposes. EAExpressions are under a detailed discussion, in terms of syntax and semantics, in the chapter 6.

- SARI allows to access event objects as XML documents and a user can use XPath for extracting data from the event object. XPath is strong in flexibly selecting elements from an XML document. Representing event objects as XML documents can be very useful for the integration with external systems, since they don't have to know any proprietary details of the event objects.

- Development of services which require complex or efficient event processing. SARI offers an API for directly accessing event objects in Java or C#. It is the most efficient way for retrieving data from the event object, since no expressions have to be parsed or interpreted.

There are various ways for accessing event objects with trade-offs for each approach. In the following, a list of typical examples are shown where different approaches for accessing event objects are needed.

- **Definition of event-triggered rules:** It should be easy to understand and to use for business users to model event-triggered business rules. Event-triggered rules typically require the definition of a set of event conditions and event patterns which can trigger an action.

- **Calculation of business metrics:** When monitoring business processes or making automated decisions with rules, it is essential to support operators and aggregation functions for calculating business metrics.
- **Filtering of event objects:** Filters are essential for event consumers in order to select only the relevant events from a stream of events.
- **Data mappings:** Event data often have to be combined with data originating from relational database systems. When inserting event data into database tables or for making database lookups, it is necessary to flexibly select attribute values from an event object.
- **Processing events with services:** Event processing tasks might also require the development of code in a programming language, such as JAVA or C#. An event model should provide a flexible and efficient API for using event objects in user-defined services. In the following, there are some typical examples for application scenarios shown for the previously discussed approaches accessing event objects:

Application Scenario	**Example**
Definition of condition for a rule which checks the total of the UMTS service usage for a particular customer.	Event access expression: Sum(ServiceUsage(Type="UMTS").Minutes) = 500
The order items of an OrderSubmitted event should be extracted and used in an XSLT transformation.	XPath expression: //OrderItems

Development of a service which processes ServiceUsage events.
The service can directly access event information such as the type of service used.
	Code in Java:
	void process(ServiceUsage eventToProcess) {
	if(eventToProcess.Type=="UMTS") {Ě}
	}

5.2.9 Comparison of Event Models

In this section a table is represented with a comparison of event type models of several existing event processing systems. The table includes six event-based systems analyzed according to their event typing concepts. The solutions that where selected consider themselves as complex event processing and event stream processing systems. The selection was done with the intention to highlight the major differences between various approaches. The comparison in the table reveals commonalities and trends in event models of event processing systems. For instances, systems making use of SQL-like syntax represent events as data tuples, which cannot be hierarchically structured in many systems. Tuple-based event models bring the drawback that more advanced typing concepts are difficult to realize and thus, they are missing in such event processing systems.

Other systems, such as Esper, make use of more object-oriented approaches for typing and representing events. This allows a more natural realization of concepts like type hierarchies. A commonality that all event processing systems has is that they use XML for structuring, typing or representing events. Clearly, XML has proven to be the best choice for structuring event data or maintaining their meta information.

5.3 Event Correlation

A key characteristic of event-based systems is that they are capable of attaching themselves to a wide range of event producing sources. Thus large amounts of different types of events and at different granularity levels are consumed by the event processing realm. Defining and managing the relationships of those events is an important and integral part of SARI. The relationship between events is called *correlation* and defines a collection of semantic rules to describe how specific events are related to each other. Often single events may not contain all the information that is required to detect specific patterns in order to optimize the businesses or trigger countermeasures on exceptional situations. Furthermore, correlated events can be used to calculate metrics out of related events or metrics out of groups of correlating events.

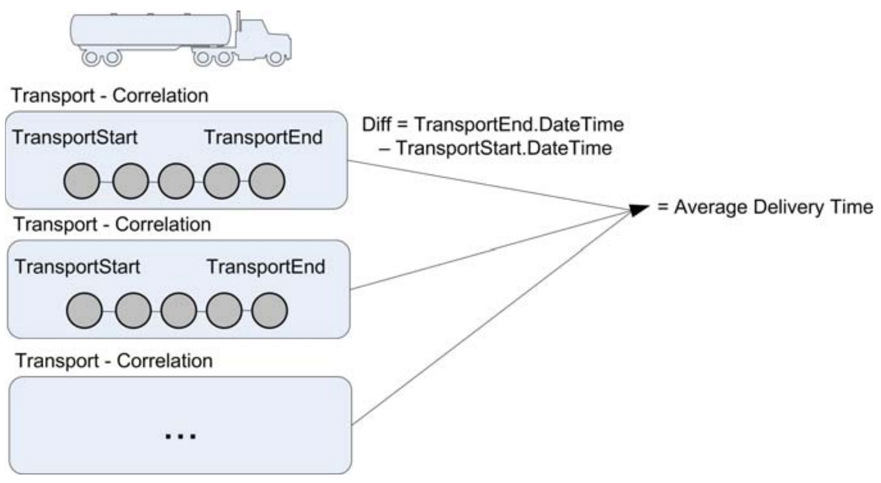


Figure 5.15: Simple Correlation Metric Calculation Example

	Event Type Model	**Attribute Model**	**Inheritance/Exheritance**
SARI	Repository for event type definitions; Globally addressable event types; Event type definition in XML; Validation through XSD;	Single-value types; Collection types; Dictionary types; Support of unknown attributes;	Inheritance supported; Exheritance supported;
Esper	Event type definition in XML; Supports structured events; Validation through XSD;	Attribute types are determined by POJOs following the JavaBean conventions; Dictionary and collection types are supported;	Inheritance supported through OO concepts in Java;
Hermes	Event type definition in XML; Validation through XSD;	XML representation that can be bound to Java classes;	Inheritance supported
Borealis	Schema defines valid tuples; Validation through XSD;	Single-value types supporting number, date and string values;	
RuleCore	Event type definition in XML; Validation through XSD;	Single-value types supporting number, date and string values;	
AMIT	Event type definition in XML; Validation through XSD;	Single-value types supporting number, date and string values;	Inheritance supported

	Advanced Typing Concepts	**Addressing**	**Event Access**	**Event Representation**

SARI	Duck Typing;	URI;	EAExpressions;
			XPath;
			API;
				Java Objects;
				.Net Objects;
				XML;

Esper	Event Aliasing	Namespace implicitly available through Java;	SQLlike syntax;
			XPath;
			API;
				Java Objects;
				XML;

HERMES		XML namespace for avoiding naming conflicts;	Programmatically;
			XPath;
				Java Objects;

Borealis			SQLlike syntax;	C++ structs;
RuleCore		Can be enforces implicitly;	XPath;	Python classes;

AMIT		Event types have an identifier;	SQLlike syntax;	Java Objects;
				XML;

For instance events could be used to calculate the average delivery time of transports (illustration in Figure 5.15). A *Transport* is a correlation consisting of several events that signalize different activities during a transport process. Each of the transport correlations contains at least a *TransportStart* and a *TransportEnd* event. The time difference between the *TransportStart* and *TransportEnd* events, of each transports, is used to calculate the average delivery time. The ability to perform such calculation operations is enabled through the technique of correlations discussed subsequently.

ESP solutions, based on the concept of continuous queries, realize correlations usually through join operations (7) similar to relational databases. This means that events are brought into relationship by defining joins over specific attributes, whereas the resulting records, defined by the projection clauses, represent the result of correlations. To preserve the correlated events it would be required to form foreign-key related data structures. This approach is limited by the nature of attribute-relationships and by the sliding time-windows of such solutions. Events, like in a transport scenario, occur often only throughout a long lasting duration. This means that maintaining correlations in such ways would become a high resources consuming task due to the complexity nature of relational operations.

The correlation technique applied in rules, event processing maps, data mining applications and queries would be a difficult task to perform. SARI applies a correlation method describe by Schiefer et al. [79] which is introduced in this section.

5.3.1 Types of Correlations

The definition of a correlation between event types is called a *correlation set*. Correlations are defined through specifying correlating attributes between event types. SARI supports two types of correlation sets *primal correlations* and *bridged correlations*. A correlation set consists of an unique name, the event types that participate in this correlation set, and the event attributes that relate to each other. In the following the correlation types will be described in detail.

5.3.1.1 Primal Correlation

A primal correlation set is the simplest correlation type and forms the basis for other correlation conjunctions between events. Events that enter the event processing realm are typed against the event object type library, holding all the event typing information, and are instantiated as event objects during runtime. An event type consists of several event attributes, which are of a specific defined type. A correlation between events can be defined by defining a relationship between the attributes of specific event types.

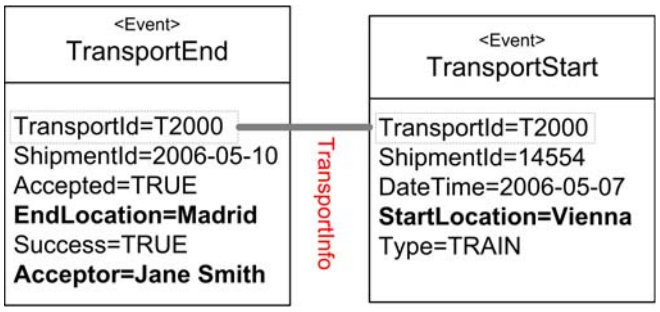


Figure 5.16: Example of a Primal Correlation

Figure 5.16 shows a relationship between *TransportEnd* and *TransportStart* events. The relationship is defined by the associated attributes *TransportId* of those two event types. A correlation set may include one or more event types which define relationships by correlation tuples. A correlation tuple defines attributes from different event types which have to match in order to correlate. A relationship may include one or more correlating attributes which are part of correlation tuples.

A primal correlation may also contain a correlation set consisting of a on correlation tuple referencing one attribute of one event type. That correlation definition is called self-correlation. For instance, if the event type *TransportEnd* has a self correlation defined on the *EndLocation*, all *TransportEnd* events with the same end locations would share a relationship. A concrete application scenario would be if a user wants to collect all outgoing transports and then wants to evaluate how many transports have failed grouped by their delivery locations.

5.3.1.2 Bridged Correlation

The primal correlation set defines direct correlational relationships between event types and their attributes. The bridged correlation extends this model by allowing to define correlations between several primal correlation. This type of correlation allows to form indirect relationships between events through defining bridging attributes between primal sets of correlations.

A bridged correlation set consists of a set of primal correlation sets that are linked together. The example in Figure 5.17 extends the previous primal correlation set example (TransportInfo), shown in Figure 5.16, with an additional correlation set (DemandToShipment). The bridged correlation allows to create a relationship between those two primal correlation sets by defining a link between the attributes ShipmentId and OrderId. This link now bridges two different primal correlation sets

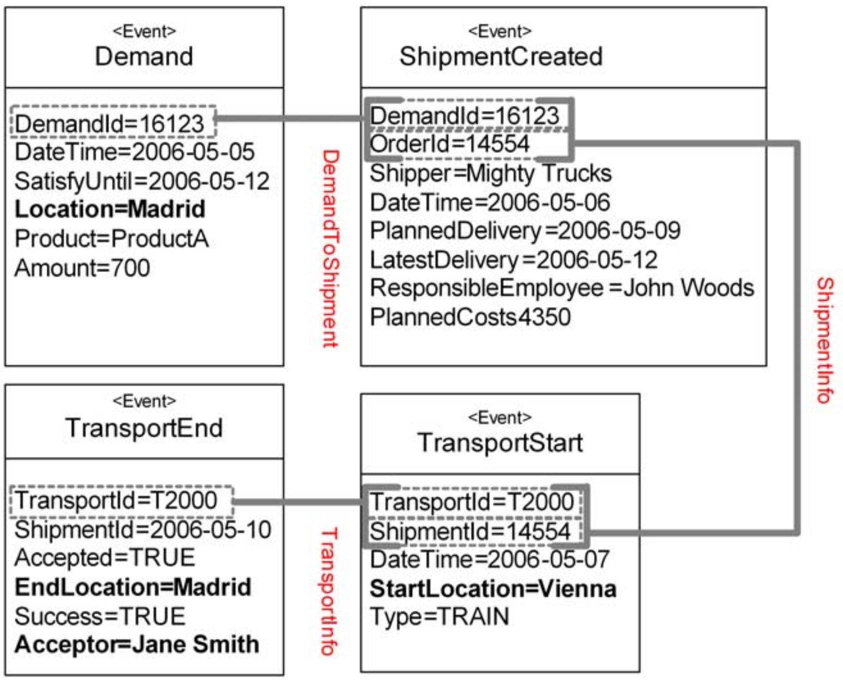


Figure 5.17: Example of a Bridged Correlation

and thus they extend the correlation scope.

5.3.2 Correlation Meta Model

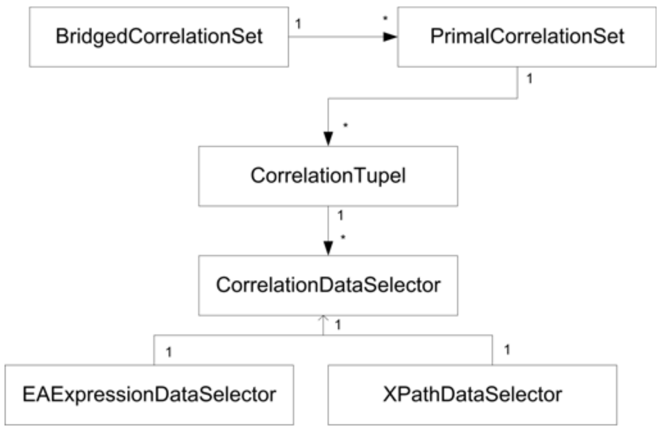


Figure 5.18: Correlation Meta Model

The correlation meta model, illustrated in Figure 5.18, is a straight forward nested structure for defining correlations for event processing model components. A *Bridged-CorrelationSet* is the main entry point for hanging on correlation definition information. Each *BridgedCorrelationSet* is described by a unique identifier and optionally through a display name. A *BridgedCorrelationSet* contains an arbitrary number, but at least one, of *PrimalCorrelationSets*. The *PrimalCorrelationSet* is identified by a unique id and can hold more than one *CorrelationTuple* whereas the *CorrelationTuple* contains a so called *CorrelationDataSelector*. The *CorrelationDataSelector* defines the event object type that is under a correlation selection. Further, the *CorrelationDataSelector* contains a query expression for selecting the correlation attribute. The attribute selection can either be done using *EAExpressions* or *XPath*.

5.3.3 Correlation Evaluation

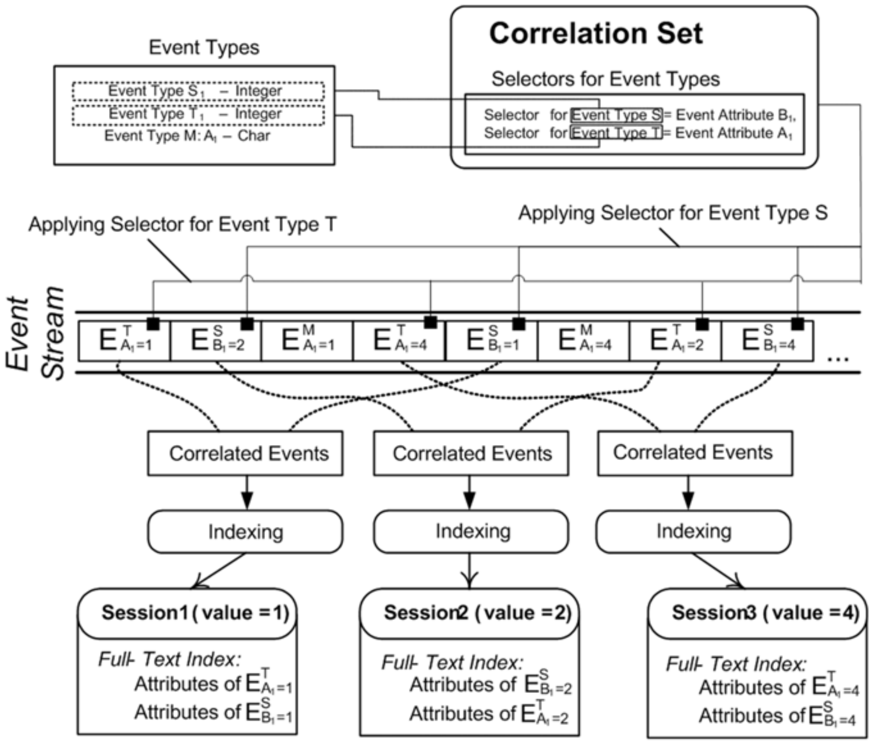


Figure 5.19: Correlation Evaluation during Runtime

The correlation evaluation technique, introduced and described in [79], that is applied during runtime uses defined correlation sets to collect attribute values of correlating events for business activities. The gathered attribute values are held in a data container which we define as correlation session. In other words, a correlation session is a container with a set of data items that exists for each relationship between events. Figure 5.19 illustrates the whole life cycle of the correlation evaluation process. On the top there is a correlation set definition (either a primal or bridged correlation) that consists of tuples of event selectors. Those selectors define the event object types and their attributes that are linked together. In this example the event object types S and T are correlated through their attributes B_1 A_1. The event object type M is not part of the shown correlation set definition.

The event stream in the middle of the picture represents the flow of events E with the types S, T and M. The events streaming through are concrete instantiations of event object types and thus the attributes B_1 A_1 contain values. An event correlation is defined by a correlation set which consists of a set of selectors for various event types.

According to [76] the correlation is described: For a given event of an event stream, an event correlation is performed successfully if

1. the event type of the given event conforms to the event type of one of the selectors of the correlation set,
2. this selector is used to extract one or more event attributes from the events, and
3. the extracted event data is used to assess the events as correlating if the attributes extracted by the responding selectors match.

In Event Cloud [76] the correlation approach described by Schiefer and McGregor [79] was extended by an indexing approach to provide an acceleration of a full text search of historical events. Therefore, correlation sessions (shown in the bottom of the figure) are coupled to an indexing approach that is continuously updated as new events are gathered by the sessions.

5.4 Event-Driven Rules

Information is critical to make well-informed decisions. This is true not only in real life, but also in computing and especially critical in several areas, such as finance, fraud detection, business intelligence or business process management. Information flows in from different sources in the form of messages or events, giving a hint of the state at a given time, such as the completion of shipping an order.

Business Process Management (BPM) systems are software solutions that support the management of the lifecycle of a business process. For the execution of business processes, many organizations are increasingly using process engines supporting standard-based process models (such as WSBPEL) to improve the efficiency of their processes and keep the testing independent from specific middleware.

A major challenge of current BPM solutions is to continuously monitor ongoing activities in a business environment [78] and to respond to business events with minimal latency.

In SARI, business situations and exceptions are modeled with sense and respond rules which have been designed to be created and modified by business users. SARI offers a user-friendly modeling interface for event-triggered rules, which allows to model rules by breaking them down into simple, understandable elements. The main advantage lies in a new model for constructing rules with a correlation model and a graph for representing business situations with a combination of event conditions and event patterns. The proposed model can be seamlessly integrated into the distributed and service-oriented event processing platform of SARI.

Related work can be divided into work on active event processing and event algebras in the active database community and work on event/action logics, updates, state processing/transitions and temporal reasoning in the knowledge representation domain.

There has been a lot of research and development concerning knowledge updates and active rules in the area of active databases and several techniques based on syntactic (e.g. triggering graphs or activation graphs [12]) and semantics analysis (e.g. [9]) of rules have been proposed to ensure termination of active rules (no cycles between rules) and confluence of update programs (always one unique outcome). The combination of deductive and active rules has been also investigated in different approaches manly based on the simulation of active rules by means of deductive rules [57]. However, in contrast to this model, these approaches often assume a very simplified operational model for active rules without complex events and ECA-related event processing.

RuleCore [82] is an event driven rule processing engine supporting Event Condition Action (ECA) rules, and providing a user interface for rule building and composite event definition.

Wu et al. [17] proposes event correlation approach with rules in the "conclusion if condition" form which are used to match incoming events often via an inference engine. Based on the results of each test, and the combination of events in the system, the rule-processing engine analyzes data until it reaches a final state.

Chen et al. [27] show an approach for rule-based event correlation. In their approach, they correlate and adapted complex/structural XML events corresponding to an XML schema. The authors describe an approach for translating hierarchical structured events into an event model which uses name-value pairs for storing event attributes.

ECA rules have been also proposed by several authors for workflow execution, e.g., [13][21][35][46]. In event-driven workflow execution, events and event-condition-action rules are the fundamental mechanisms for defining and enforcing workflow logic. Processing entities enact workflows by reacting to and generating new events. The foundation on events facilitates the integration of processing entities into coherent systems. Some of these systems [13][35] use composite events to detect complex workflow situations. EVE [46] is a system using ECA rules for workflow management addressing the problem of distributed event-based workflow execution.

5.4.1 Event-Driven Rules

Sense and respond rules allow to describe and discover business situations and can automatically trigger responses such as generating early warnings, preventing damage, loss or excessive cost, exploiting time-critical business opportunities, or adapting business systems with minimal latency. The key requirements of sense and respond rules can be summarized as follows:

- **Event-triggered Rule Evaluation:** Sense and respond rules enable companies to monitor their business, IT and organizational processes in real-time, and respond to exceptions and capitalize on time-sensitive business opportunities as soon as new events occur within the business environment. In other words, the evaluation of sense and respond rules is triggered by events delivering the most recent state and information from the business environment.

- **User-friendly Rule Modeling:** Sense and respond rules support the graphic modeling of decision-making scenarios. Decision trees proved to be very understandable for human beings. For modeling business situations, sense and respond rules use decision graphs which are an extension of decision trees for representing rules, thereby enhancing the understandability and expressiveness of rules, and shortening the learning curve for users.

- **Building Complex Rules with Divide and Conquer:** Sense and respond rules can break down complex business situations in simple understandable conditions, which can be combined with eachother for composing more complex conditions. The input of sense and respond rules are events and also the output are also so called response events, which are raised when a rule fires. The fired events can be used as input for other (or even the same) rule for further evaluation, thereby effectively combining multiple rules.

- **Event Pattern Recognition:** Event patterns are discovered when an event or multiple events occur that match the pattern's definition. Sense and respond rules allow to combine one or more event pattern with arbitrary event conditions in order to describe complex business situations.
- **Adaptability:** Due to the graphical model and modular approach for constructing rules, sense and respond rules can be easily adapted to business changes. New event conditions or event patterns can be added or removed from the rule model in order to model changing business situations.
- **Service-oriented Rule Processing:** Sense and respond rules are executed by event services, which supply the rule engine with events and process the evaluation result. Event services can run distributed on multiple machines and facilitate the integration with external systems.

5.4.1.1 Definitions of Sense and Respond

Sense and respond rules separate multiple aspects for the definition of rules which are specified and modeled separately. These three aspects are as follows:

- **Correlation.** The definition of relationships and dependencies between events that are relevant for the rule processing are performed declaratively with correlation sets. With correlation sets, a rule engine is able to construct sequences of events that are applied to the condition defined in the rule. A business activity spanning a period of time can be represented by the interval between two or more events. For example, a transport might have a *TransportStart* and *TransportEnd* event-pair. Similarly, a shipment could be represented by the events *ShipmentCreated*, *ShipmentDelivered* and multiple transport event-pairs. A correlation set allows to associate such events by using the events context data (e.g. *ShipmentID*, or *TransportID*).
- **Event/Condition/Action (ECA) Model.** Sense and respond rules use a graphical model for describing constraints of events for business situations. ECA rules automatically perform actions in response to events provided stated conditions hold. The actions of sense and respond rules generate response events, which can be used triggering business activities or evaluating further rules.
- **Event Patterns.** Event patterns complement the ECA sets and are able to match temporal event sequences. They allow to describe business situations where the order and occurrences of events are relevant, such as the discovery of missing events (e.g. a door is opened without closing it on time).

5.4.1.2 Meta Model

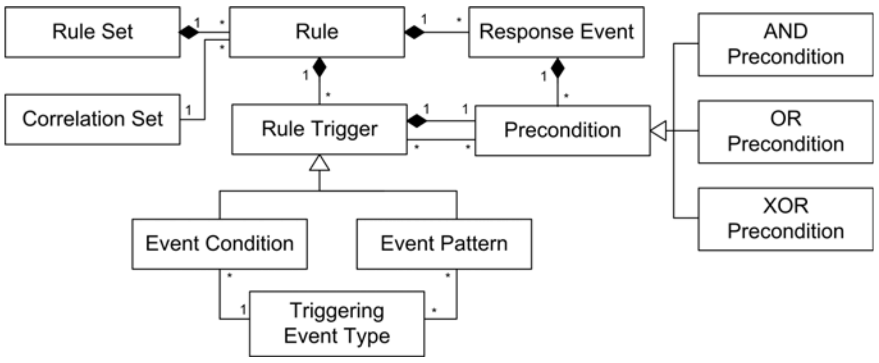


Figure 5.20: Meta Model for Sense and Respond Rules

Sense and respond rules are organized in rule sets and allow to construct decision scenarios, which use *event conditions* and *event patterns* for triggering *response events*. Event conditions and event patterns can be arbitrarily combined with logical operators in order to model complex situations. Response events are generated when event conditions or event patterns evaluate to true. Figure 5.20 shows the meta model of sense and respond rules.

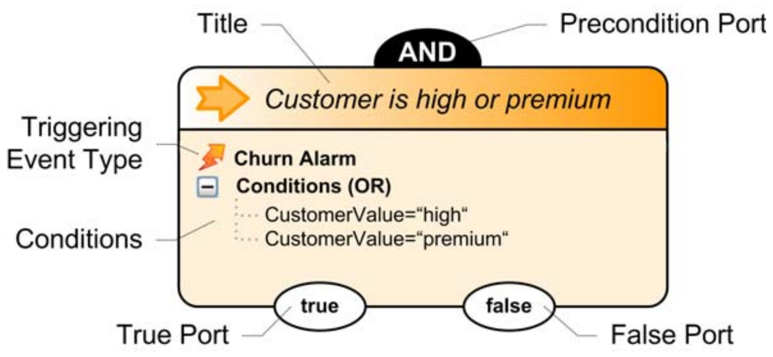


Figure 5.21: Event Conditions

5.4.1.2.1 Event Conditions. Event conditions have a triggering event and a list of Boolean expressions that are evaluated when the triggering event was received. Figure 5.21 shows an example for a "Churn Alarm" event with a list of two conditions which are combined with a logical OR operator. The event condition evaluates to true if the customer value is high or premium. When evaluating the rule, event conditions do not require to maintain any kind of state in runtime.

Event conditions have the following elements:

- *Title:* Defines in natural language a summary of the conditions.
- *Triggering Event Type:* Defines the type of events which can trigger the evaluation of the conditions.
- *Conditions:* Defines a list of conditions for the trigger event. For the definition of a condition, SARI uses event access expressions, which are defined in an own language that was designed for easy use by business users. The language used for defining the conditions is called EAExpression and is under a discussion in the section 6. The conditions can be combined with the logical operators AND, OR, or XOR.
- *True Port:* This port can be used to connect other event conditions, event patterns, and/or response events in case the event conditions evaluate to true.
- *False Port:* This port can be used to connect other event conditions, event patterns, and/or response events in case the event conditions evaluate to false.
- *Precondition Port:* This port can be used to connect other event conditions and event patterns, which should be used as a precondition for the current event conditions box. Preconditions can be combined with the logical operators AND or OR.

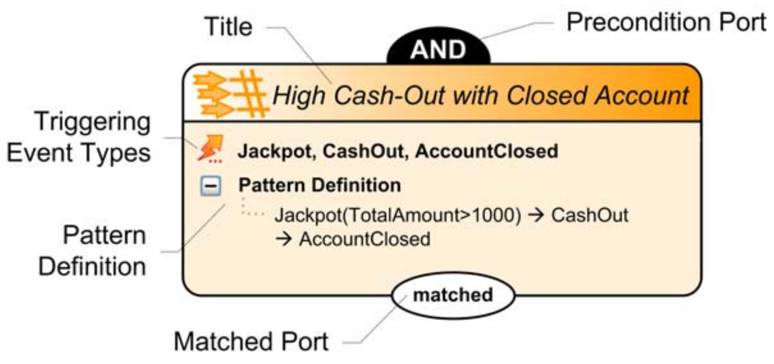

Title
Precondition Port
AND
High Cash-Out with Closed Account
Triggering Event Types
Jackpot, CashOut, AccountClosed
Pattern Definition
Jackpot(TotalAmount>1000) → CashOut → AccountClosed
Pattern Definition
matched
Matched Port

Figure 5.22: Event Pattern

5.4.1.2.2 Event patterns. Event patterns describe business situations which have to be detected from a series of events that occur over a period of time. Event patterns require state machines [1][30] for maintaining internal state information in order to determine whether an event pattern has been matched in an event stream. Examples for event patterns are the detection of temporal event sequences [25] as well as the discovery of missing events. A key characteristic for event patterns in sense and respond rules is that they are time-based, meaning that they maintain their own internal state. An event pattern only has a "matched" state as an outcome in the case where the event pattern has been matched.

Figure 5.22 shows an example for an event pattern which matches situations of users who win a large jackpot and close their account after cashing out their winnings. During the pattern matching in runtime, the rule engine has to maintain state information on which events have been processed for a particular user.

Event patterns have the following elements:

- *Title:* Defines in natural language a summary of the event pattern.
- *Triggering Event Types:* Defines a list of types of events which can trigger the evaluation of the conditions.
- *Pattern Definition:* Defines a pattern expression which uses the triggering events for the pattern matching. For the event pattern definition, SARI uses the event pattern language (EPL) from the Esper stream engine [30].
- *Matched Port:* This port can be used to connect event conditions, other event patterns, and/or response events in case the event patterns evaluate to true.
- *Precondition Port:* This port can be used to connect event conditions or other event patterns that should be used as precondition for the current event pattern. Preconditions can be combined with the logical operators AND or OR.

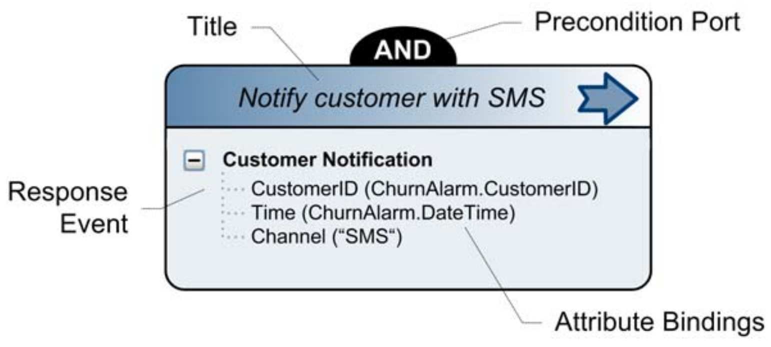


Figure 5.23: Response Events

5.4.1.2.3 Response Events. Response events represent actions which should be initiated when event conditions evaluate to true and/or event patterns match. Response events can be arbitrary events with attributes that can be bound to attributes of triggering events. In other words, response events can capture event data from triggering events in order to maintain context information for initiating an appropriate action.

Figure 5.23 shows a response event for notifying a customer with an SMS message. Two attributes of the response events are bound to the triggering event, and the attribute for the channel is set with a constant value. A response event can be connected to existing event conditions or patterns which become that precondition for firing the response event.

Response events have the following elements:

- *Title:* Defines in natural language what the response event is about

- *Response Event:* Shows the name and attributes of the response service. The root node shows the name of the event type which has event attributes as the child nodes.

- *Attribute Bindings:* Each event attribute shows in brackets binding information on how the attribute value should be calculated or set. This can be constants, calculation expression, or an attribute binding to a triggering event.

- *Precondition Port:* This port can be used to connect event conditions or other event patterns, which should be used as a precondition for the current event pattern. Preconditions can be combined with the logical operators AND or OR.

5.4.1.3 Rule Example

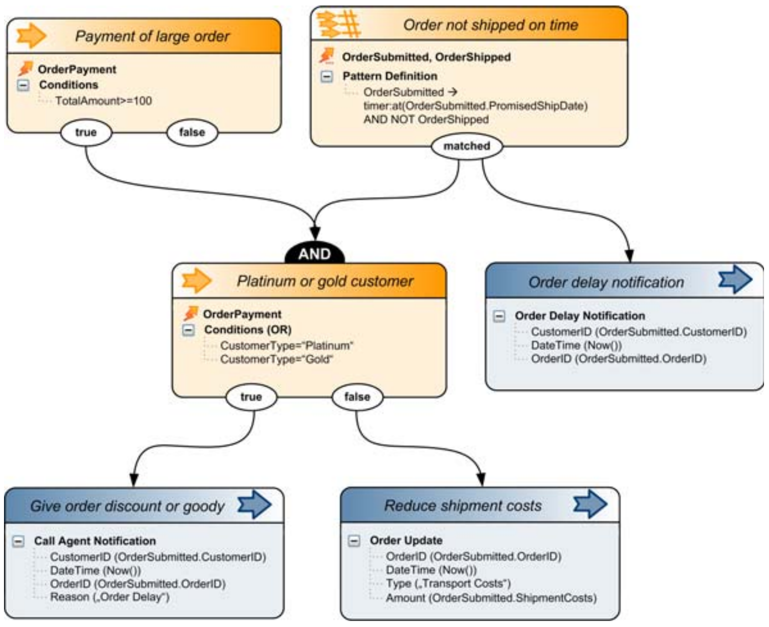


Figure 5.24: Sense and respond rule for responding to order delays

In the following, sense and respond rules are shown in context of an example for monitoring order delays. For this example, it is assumed, that the rule should discover orders which are not shipped on time. For delayed orders, the system should always send a delay notification to the customer. If the customer has been already charged for the ordered products and it was a larger order, the customer should receive a compensation for the inconvenience of the order delay. If the customer is a premium customer, he or she should be called by a call agent, who offers a discount for the order or, as an alternative, a goody such as some extra item which will be added to the order. On the other hand, non-premium customers should get a free shipment for the order.

When modeling a rule for the above-stated business problem, first the events have to be identified that which will be evaluated for the rule processing. For example, the following event types are considered:

- *Order Submitted:* Triggered when a customer places an order.
- *Order Shipped:* Signals that an order is shipped.
- *Order Payment:* A customer was charged for the ordered items.

Before defining a rule, the identified event types need to be linked, so that the rule service is able to correlate the events during the rule processing. Figure 5.16 in section 5.3 shows a correlation set which declaratively defines dependencies between event types that are relevant for the rule processing.

Figure 5.24 show a sense and respond rule for the above-stated business scenario. An event pattern box is used to match a situation for detecting orders which are not shipped on time. The event pattern requires the events *OrderSubmitted* and *OrderShipped* for discovering the order delays. In the case that an event pattern was matched, a response event *OrderDelayNotification* is generated. The response event includes a set of attributes with bindings to the source events (e.g. *CustomerID*, *OrderID*) and a function for setting the current date.

The event condition defined in the box on the top left corner checks whether the customer has already paid for the order. In the case of an order amount larger than € 100,–, the event condition will evaluate to true. The event condition box in the middle checks whether the customer is a gold or platinum customer. The box has an AND precondition, with connections to the previously discussed event condition and event pattern. The precondition will become part of the stated condition meaning that the middle condition only evaluates to true 1) the order has not been shipped on time, 2) the order has been paid and the order total was larger than € 100,–, and 3) the customer type was gold or platinum. If all conditions and preconditions evaluate to true, a response event is generated which notifies the call agent for giving the customer an order discount or a goody. If the customer type is not gold or platinum, a response event is generated for reducing the shipment costs from the order.

5.4.2 Comparison and Key Benefits

Sense and respond rules offer a way for business users to define and manage the response to typical patterns of business events. A key advantage of sense and respond rules is that they allow to graphically model a comprehensive set of event conditions and event patterns without using nested or complex expressions. In contrast, SQL-based system for querying event streams require technical knowledge for specifying and changing event queries which makes them difficult to use for a wider range of users.

Preconditions allow the combination of multiple event conditions and event pattern, thereby extending the scope of event conditions and event patterns. By combining event conditions and event patterns every element of the rule model can cover a single concern.

Sense and respond rules combine event conditions and event patterns which can be used to define IF-THEN-ELSE decisions. Event conditions can be checked for a true or false evaluation, which facilitates the implementation of "otherwise" situations. In other words, with sense and respond rules the outcome of evaluations for event conditions and event patterns can be used for multiple decision scenarios. This significantly facilitates the modeling of complex business situations with many dependent event conditions and event patterns.

By describing correlation aspects in a separate model, the definition of event conditions and patterns are simplified. Correlation sets capture the relationships between events and can also be defined with a graphical model.

The service-oriented event processing system of SARI allows to flexibly link a rule service, processing sense and respond rules with other services. Event services are executed in parallel and controlled by the system. Services can be used to prepare the data for the rule processing as well as to process the response events generated by the rule service. The input and output for sense and respond rules are events which are delivered and processed by event services. Sense and respond rules, therefore, allow an easy integration within a service-oriented system environment.

However, the SARI system also has a drawback compared to SQL-based approaches for event stream processing. Using SQL for event stream queries allows to seamlessly integrate relational database systems by joining database tables with events that stream into the system. SARI requires services for preparing event data for the rule processing.

5.4.3 Service-Oriented Rule Processing

The SARI system uses an event processing model (EPM) for modeling event-driven processes. An EPM allows to integrate multiple services and adapters which can be used to implement event-driven processes. Dependent on the requirements and the business problem, the event services and adapters can be flexibly conjoined or disconnected. Links between the components and services represent a flow of events from one service to the next.

The following issues are defined with the EPM:

- Structure for the processed events and data
- Configuration of services and adapters for processing steps, including their input and output parameters
- Interfaces to external systems for receiving data (Sense) and also for responding by executing business transactions (Respond)

- Data transformations, data analysis and persistence

A rule service is part of an EPM and can be configured with rule sets and linked with other services. EPMs allow to model which events should be processed by the rule service and how the response events should be forwarded to other event services. Figure 5.25 shows an EPM for the previous example. Data is collected and received from adapters which forward events to event services that consume them. Initially the events are enriched in order to prepare the event data for the rule processing. A typical example would be the attachment of information about the customer value (e.g. whether the customer is a platinum or gold customer). The rule service processes the enriched events according to the sense and respond rules and generates response events when a rule fires. The fired response events are published on the output port of the rule service and can be forwarded to other services. As shown in Figure 5.25, the response events are sent to a service for sending notifications to a call agent, or to services which transmit order delay notifications and order updates back to the order management system. Sense and respond rules are stored and managed with a rule repository. SARI includes authoring and management tools which can be used by business users for graphically defining and modifying rule sets as introduced in the previous sections.

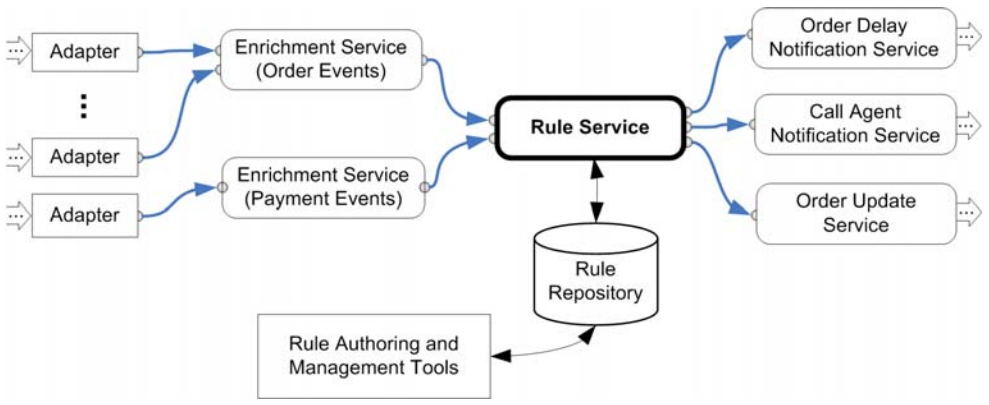

Adapter
Adapter
Adapter
Enrichment Service (Order Events)
Enrichment Service (Payment Events)
Rule Service
Order Delay Notification Service
Call Agent Notification Service
Order Update Service
Rule Repository
Rule Authoring and Management Tools

Figure 5.25: Event Processing Model with Rule Service

5.4.3.1 Rule Evaluation

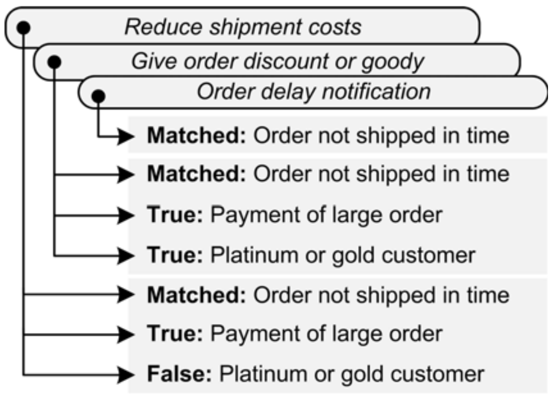

Reduce shipment costs
Give order discount or goody
Order delay notification
Matched: Order not shipped in time
Matched: Order not shipped in time
True: Payment of large order
True: Platinum or gold customer
Matched: Order not shipped in time
True: Payment of large order
False: Platinum or gold customer

Figure 5.26: Dependencies for Response Events

During runtime, a rule service automatically correlates events emitted to the SARI system. Correlated event data is managed with correlation sessions [79], which are automatically activated before a triggering event is used for evaluating event conditions or patterns. Correlation sessions can be persistent and are used to maintain the current rule state. Event conditions and patterns can use the captured event data for accessing data of correlated events which has been previously processed.

The rule service uses a list of precondition dependencies for each response event which is created before the rule processing starts. Figure 5.26 shows the dependency list from the previously shown example. If the current evaluation state of a rule matches a state represented in the dependency list, a response event will be fired.

Please note, the rule service does not consider the order of the processed events during the rule evaluation. In other words, it does not matter in this example in which order the event conditions or patterns are evaluated. The current evaluation state of the rule is always used to match with the dependency lists of response events.

5.5 Data Management

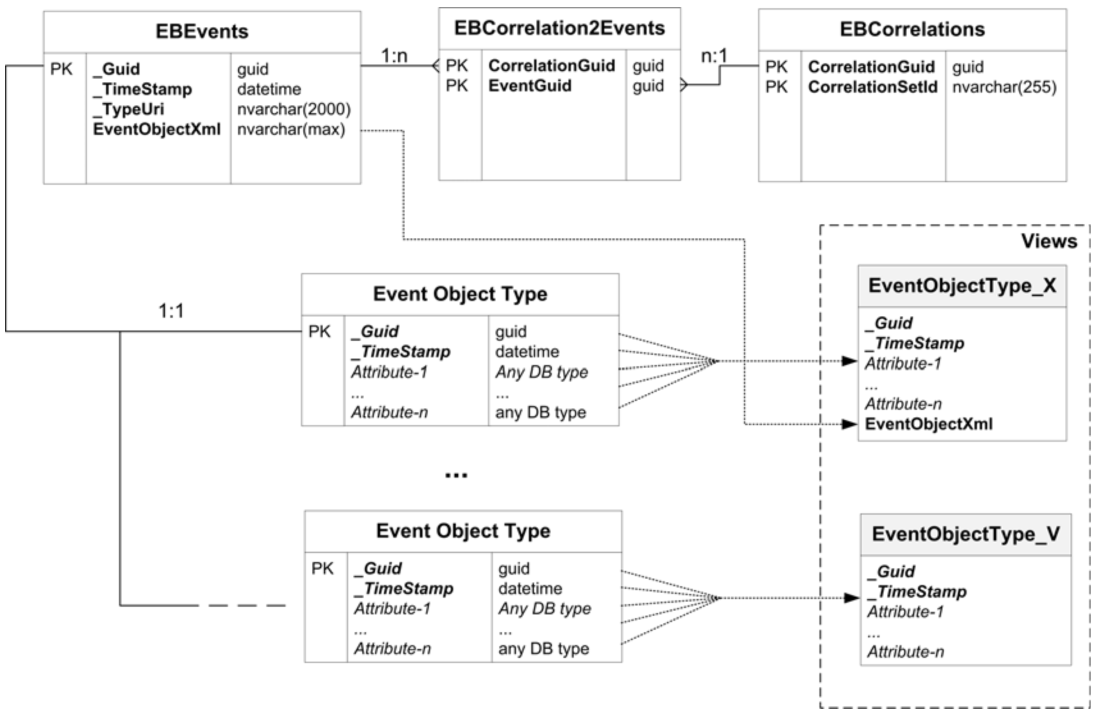


Figure 5.27: EventBase Data Model

This section pays attention to how events and their correlations are processed on a conceptual level to persists them to the underlying data structures and further how the events are maintained for further access. This topic is highly relevant for SARI-SQL as it is the data repository that is used by the SARI-SQL engine to retrieve the data that is requested by the given query.

Figure 5.27 shows the Entity-Relationship (ER) diagram of the underlying data structure of the EventBase for managing the events and their correlations. The centerpiece is formed by the event object type relations shown in the middle of the figure. Referring back to the meta model shown in Figure 5.4 of event object types in the Section 5.2.1, each event object type consists of one or more attributes. An attribute type can be either a single-value type, collection/dictionary type or another nested event object type. Single value types correspond to a ordinary runtime types such as Integer, String and so forth. During execution runtime of the EventBase relations for all event object types maintained in the event object type library are created. Single value type attributes are represented by relational-attributes with a database type corresponding to the event object attribute type. In contemporary RDBMS there is for every runtime type a corresponding relational attribute type available. For each event object type there is a unique identifier and a timestamp defined during execution runtime.

For nested event object types and collection/dictionary types there are no additional relational-attributes created. The representation of such nested types is solved by the *EventObjectXml* attribute in the relation *EBEvents* which also contains a unique type resource identifier with a reference to the corresponding EventBase repository that it belongs to. The *EBEvents* contains a 1:1 mapping to the corresponding event object type over the unique identifier *_Guid* of the Event Object. The relational-attribute *EventObjectXml* contains the raw XML representation of the event objects including the nested types like dictionaries, event object type attributes or unknown attributes.

The relation *EBCorrelations* represents correlation sets and consists of a unique identifier *CorrelationGuid* and a *CorrelationSetId* containing the correlation identifier (e.g. name). The two relations *EBCorrelations* and *CorrelationGuid* form a n:m relationship which is resolved by *EBCorrelation2Events*.

For example, let's consider *TransportStart* and a *TransportEnd* events that correlated over an *OrderId* attribute with eachother. During event processing runtime the event objects of both events are inserted into the *EBEvents* relations and in parallel into their corresponding event object type relation for the *TransportStart* and *TransportEnd* events. Correlated *TransportStart* and *TransportEnd* events are brought into relationship through an entry in the *EBCorrelation2Events* relation whereas the events in a specific correlation can be retrieved through the *EBCorrelations* by querying for the *CorrelationSetId*.

Now that the event type specific relations don't contain the whole event object data, like nested types or any other specific header attributes, the extended *EventObjectType_X* View is introduced. Such a view exists for every event object type maintained in the event object type Library. The view contains every “flat“ attribute (e.g. single value type attribute) of an event and in addition it offers the *EventObjectTypeXml*, including the nested types, for access. Basically, the *EventObjectType_X* can be accessed to retrieve the full information of an event.

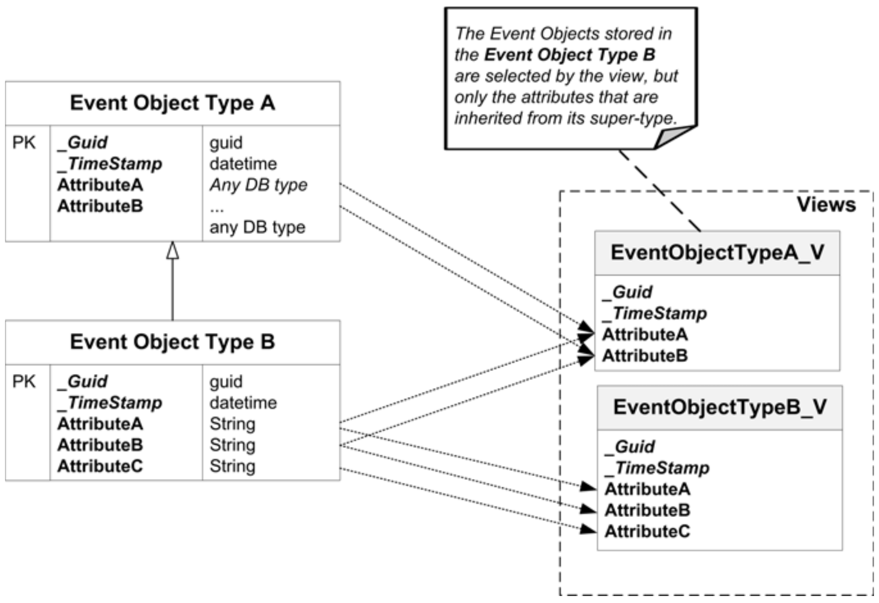


Figure 5.28: Event Object Type Inheritance Data Management

The last issue of this topic is the management of event object type inheritances. Advanced event typing concepts were discussed in detail in Section 5.2.1. Event object types can be specialized by inheriting attributes from parent types similar to object-oriented programming languages. Figure 5.28 shows an illustration of two event object types where the *Event Object Type B* is inherited from *Event Object Type A*. Originally, the *Event Object Type A* contains the two attributes (*AttributeA* and *AttributeB*). The *Event Object Type B* introduces additionally the *AttributeC*. In case of inherited types the attributes of the super-types are duplicated in the relations of the specialized types. If the view of the super-types are accessed, every event object is included from all derived types, but with the difference that only the attributes of the super-types are available.

5.6 Summary

This chapter introduced the CEP solution SARI with a special attention to its extension the Event-Base. SARI allows observing relevant business events to identify exceptional situations, indicate opportunities or problems combined with low latency times in decision making for supportive or counter measures. The Event-Base on the other hand provides an efficient up-to-date operational storage together with retrieval mechanisms for business events for analytical as well as operational purposes without the costly data staging processes known from established data warehousing solution.

Event models have a major impact on the flexibility and usability. Furthermore the underlying event model and its various event typing concepts were discussed intensively as the design and the nature of the event model strongly constraints the capabilities of event processing query languages. Therefore a dedicated section introduced concepts and approaches for representing, structuring and typing event data and introduced event models of existing event-based solutions. Several concepts of organizing event models, basic typing concepts for structuring event data, as well as more advanced typing concepts such as inheritance, exheritance and dynamic type inferencing were discussed. The various typing concepts were illustrated with the event-based system SARI and compared with existing event-based systems. So for, ESP solutions provide SQL-like query languages, because they use as the underlying event model representation relational tables.

A special attention was also set on event-driven sense and respond rules, which can be used to model trees with event actions within the event processing model. Such rules allow to compose business situations with event conditions and event patterns which can be arbitrary combined and trigger respond events when a rule fires. Sense and respond rules can be graphically modeled which makes it easier for business users to adapt rules for business changes. In runtime, the rules are processed by event services running on multiple machines and which can be seamlessly integrated with other existing services. However, at first glance those rules might not have fit into the concept of this chapter as they are a special aspect of the event processing model of SARI. But looking closer, it reveals that they are using the domain-specific language EAExpressions for accessing and performing operations on events, which is a subset of the query language SARI-SQL. As a consequence the extension of this rule component with SARI-SQL would allow to integrate historical event data, whole event correlations or metrics into the decision process.

A section focused on the data management in the Event-Base described how events and their correlations are processed efficiently on a conceptual level to persist them to the underlying data structures and further how the events are maintained for further access. The data management is an important component of the Event-Base as it is the data repository that is used by the SARI-SQL engine to retrieve the data that is requested by the given query. Several issues were discussed on how events and their correlations are processed and persisted to the underlying data structures and further how the events are maintained for access by SARI-SQL.

The event processing concepts of SARI and the Event-Base were explained with the help of a use-case scenario for preventing and managing fraud situations in the online betting domain. The scenario follows event-driven architecture principles and applies event-driven rules for discovering and responding to fraud patterns. This use-case also revealed the capability of the introduced event-based system to enable real-time fraud detection and providing tools to domain experts to model and define fraud prevention requirements.

6 SARI-SQL Query Language

This chapter introduces the two languages EAExpression and SARI-SQL for event-based systems. Both languages belong to the group of so called domain-specific languages (DSL), which addresses the specific nature and the problem domain of event-based and in particular complex event processing systems. Before discussing the details of the languages subsequently in this chapter, it is necessary to clarify some background on the nature of such languages. This is a vital information that provides the reader insights why it is necessary to design such languages, especially why it is an important feature for event processing systems.

EAExpression and SARI-SQL are two languages independent from each other. However, SARI-SQL includes EAExpressions in order to be able to express the access of events in its queries.

EAExpression is an expressive and easy to understand language for accessing events, both during the design phase of event processing applications and also during runtime to perform evaluations on events. The difference is, that during the design phase event object types, known to the event processing realm, are under evaluation. The event object types define the data structure of events including their attributes and their types. This information is stored in event object type libraries of the corresponding Event-Bases. The EAExpression processor is capable of using library maintained event type information to validate a given expressions according to the accessed event types.

The EAExpression language pays a special attention to the nature of events and their specific characteristics. The access language is tightly coupled to SARIs underlying event object type model (compare with Section 5.2) and during runtime to the evaluation of expressions on event objects. However, the syntax and semantics of the language can be decoupled completely from the underlying models with certain efforts. EAExpressions play an important role in the event processing models of SARI introduced in Section 5.1. Possibly the most important application is in the event-driven rules described in Section 5.4. EAExpressions are applied in the conditions, the components that trigger on specific event object types, as conditional evaluations. So for instance EAExpressions can be used to check attributes for some conditions. In response events (compare with Section 5.4) the EAExpressions are used to map event attributes from one event object to another event object that is emitted due to an evaluation state in the event driven rules. Further, EAExpressions can be used to define filters in event processing maps (compare with Section 5.1.3). There is also the concept of modifiers in event processing maps where EAExpressions can be used to define conditions on events. If they apply, the attributes of those events can be modified (e.g. recalculated for instance) with the help of EAExpressions.

SARI-SQL on the other hand is a query language that can be used to retrieve events, correlations, metrics and scoring information from the Event-Base. The language is from a syntactical point of view comparable to ANSI-SQL and is also a declarative language. Special attention was paid to extend the concepts of ANSI-SQL by special event processing related concepts, such as providing an easy access to correlated events without breaking too much with the syntax-style of ANSI-SQL. This is because SARI-SQL is aimed on one hand at power users applying queries through APIs in their programs and on the other hand at domain experts (e.g. business users) that try to retrieve knowledge from the Event-Base. As SQL is an industry wide standard and is part of many business related applications, a lot of non-technical people know the language. Therefore, SARI-SQL might be familiar at the first glance. The special capabilities and extensions that are offered can be found very intuitively.

Both of the languages EAExpressions as well as SARI-SQL can be allocated to the group of domain-specific languages. According to the comprehensive annotated bibliography on Domain-Specific Languages [88] these class of languages can be defined as:

> *A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.*[88]

Domain-specific languages have usually a declarative nature, whereas their focus is set on providing expressive language constructs that can be applied to the problem domain that they are addressing. Several considerations have been taken into account to create a more mighty event processing language instead of “just” designing an access language for events. Usually DSLs are realized through a framework that allows to translate the DSL instructions into another form of library operations of existing programming languages. These DSL languages consist of a lexical scanner, a parser component that builds up an abstract syntax tree (AST) and an AST walker component for evaluating (e.g. executing) the DSL.

According to [88] the advantage of DSL approaches are:

- DSLs allow to create expressions that represent abstractions of a specific problem domain. So for instance EAExpressions create an abstraction of the event type model. Other possibilities could be the use of XPath or a programmatic access to event objects. However, these access forms completely ignore the metaphor of events and thus they create an overhead to the developers and users that is not relevant to the access of events.

- DSLs allow easy insight for domain experts due to the level of abstraction. SARI-SQL for instance allows to easily access correlations of events by also defining several constraints. The results can then be used by domain experts to analyze and gain insight of the actual problems.

- DSLs increase the productivity. Both SARI-SQL and EAExpression encapsulate a lot of overhead and create an abstraction layer over events, their relationships and the internal data structures. Therefore, the user of these languages can concentrate on only expressing the required results instead of putting effort into making the "things run".

However, the drawbacks can be according to [88] following points:

- Designing, implementing and maintaining DSL languages is an expensive task in terms of development time. The benefit of creating such languages must be evaluated against the costs of such undertakings.

- During the design phase of SARI-SQL and EAExpressions a special attention was paid at making them very user-friendly. However it is fact that there is some effort required for training purposes or at least time is required that the user or developer works himself into the languages.

- Sometimes the level of abstraction and the expressiveness of DSL languages is a trade-off between normal hand-crafted code. So for instance DSLs might create round trips during execution that can be avoided if the underlying code is written by hand. However, sometimes it can also be the opposite. So for instance SQL has optimizers in place that take the user created statements, decompose them and create an optimal execution plan for the query.

6.1 General note on the Syntax Definition

The EAExpression and the SARI-SQL syntax is defined with the help of a modified EBNF (Extended Backus-Naur-Form) notation and later on was translated into

ANTLRs grammar syntax during the implementation phase. ANTLR (ANother Tool for Language Recognition) *is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages. ANTLR provides excellent support for tree construction, tree walking, translation, error recovery, and error reporting.*[5]. EBNF is a formal meta-syntax for representing context-free grammars (Type-2 grammar according to the Chomsky Hierarchy) and is a common formal notation for computer language grammars. The original notation BNF was restricted by the limitation that it is not possible to express compact recurring elements in a grammar.

The syntactical rules for the languages introduced in this chapter consist of a rule-header, a non-terminal symbol, followed by a list of sequences of terminal or non-terminal symbols. Symbols are non-terminal if they don't occur in a rule-header. The rule-header and the rule-body is separated by a ⇒ symbol.

Address ⇒	*Recipient Street ZIP*
Recipient ⇒	*FirstName LastName*
	∨ *LastName FirstName* ∨ *FirstName (LastName)**
StreetName ⇒	*Name HouseNumber*
ZIP ⇒	*Country PostalCode*

The example above describes rules for creating a postal address. The entry rule is the *Address* which defines a rule containing substitution rules for a *Recipient*, a *StreetName* and a *ZIP* postal-code. Each of the three symbols are non-terminals as they occur in rule-headers. The next rule, *Recipient* defines the naming syntax choices. In this rule it is possible to swap the first name and the last name of a recipient. The choices are separated by ∨ symbol. Further the last choice in the rule encapsulates the *LastName* into braces followed by the * symbol. This star indicates that there can arbitrarily any number of last names. This is of course very simplified as in reality the creation of another rule would be required and further double names are usually separated by a "-". However for the sake of simplicity deeper considerations were not taken into account in this particular example. The last two rules are straight forward. The *StreetName* defines the syntactical rule of a street name and a house number and the *ZIP* rule requires a country and a postal code number.

In addition to the * operator there are also two other operators in use to encapsulate rule entities.

- The * operator says that an expression in the preceding braces can occur arbitrarily often.

- The *+* operator defines that the expression must occur at least once.
- The *?* operator says that an expression can either occur once or not at all.

Throughout this chapter the EAExpression and SARI-SQL syntax will be introduced step-by-step in a top down style and illustrated with example.

6.2 EAExpression

6.2.1 EAExpression Syntax Definition

The root entry point of the EAExpression syntax is a boolean expression (*boolExpr*) and defines boolean operators that allow to combine arbitrary expressions. A boolean expression is at the highest level of the operator binding hierarchy of expressions, which means that a boolean expression has the lightest binding characteristic of all other operator types. The deeper the syntax rules go the more stronger the binding is. So for instance a plus binds two expressions stronger than a boolean operator such as an AND.

eaExpression ⇒ *boolExpr*

The preceding section will explain in detail the syntax of EAExpressions and its capabilities step-by-step. The Section 9.1 in the appendix contains a complete syntax definition including syntax diagrams for a better understanding. The language concepts are explained with examples based on an event object type definition illustrated in Figure 6.1.

6.2.1.1 Accessing Event Object Types

Before digging too deep into the syntax hierarchy it is essential to understand the lowest constructs from which outgoing higher levels of language aggregates/constructs can be formed. Those lower constructs allow the user to access event object types and their specific properties.

The event model concepts of SARI were introduced in the Section 5.2 in the previous chapter 5. The typing concept is basically a definition of event types with a detailed description of the nature and structure for the events. The description model of an event is called event object type whereas events are instantiated during runtime as event objects which must be valid to a defined event object type. The event typing model supports three types of attributes. The single-value types represent attributes which have a runtime type such as a string, character, numeric and Boolean values. Collection types contain lists of values either corresponding to runtime types or to event object types. In either way, the collection values have to be typed. Dictionary

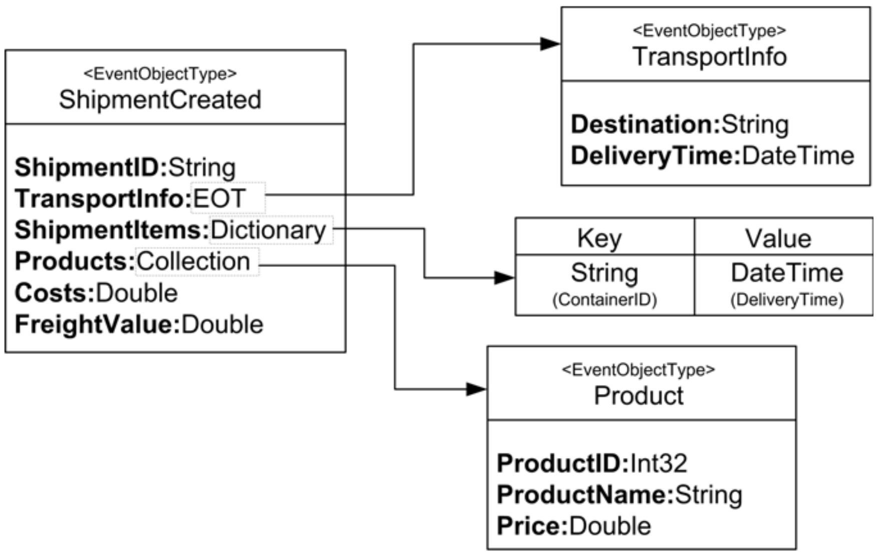


Figure 6.1: Event Object Type Example for the EAExpressions

Types contain key-value pairs where the key is the accessor for the list element and corresponds to a a runtime type. The value part is either a runtime type or an event object type.

$$
\begin{aligned}
\textbf{\textit{dotExpr}} &\Rightarrow \textit{eventAtom (DOT dotExpr)?} \vee \\
&\quad \textit{eventExprSpecial} \vee \\
&\quad \textit{functionExpr} \\
\textbf{\textit{eventExpr}} &\Rightarrow \textit{eventAtom (DOT dotExpr)?} \vee \\
&\quad \textit{eventExprSpecial} \\
\textbf{\textit{eventAtom}} &\Rightarrow \textit{IDENTIFIER}
\end{aligned}
$$

Event Object Types can be accessed by EAExpressions through a dot notation called *dotExpr* and *eventExpr* in the syntax rules. The difference between those two rules will be explained in Section 6.2.1.5. A dot is normally a separator between event object types and their attributes. The first rule definition *eventAtom (DOT dotExpr)?* allows to define an arbitrarily long combination of dot notations. The *eventAtom* itself is just an identifier in terms of a syntax. During the language evaluation this *eventAtom* must correspond to a valid event object type identifier. An event object type however is also addressable through a complete namespace which can be expressed through a set of dots (compare with Section 5.2.7). The following example shows a simple access of attributes of the *ShipmentCreated* event object type:

Example:
a) *ShipmentCreated.Costs*
b) *ShipmentCreated.FreightValue*

To access attributes of event object types it is not always necessary to write so called *full-qualified attributes* that consist of the form *eventAtom (DOT dotExpr)?* (i.e. EOT.Attribute). In case that there is only one event object type available it is possible to directly access the attributes without defining the event object type identifier. Let's consider the example where we have only one *ShipmentCreated* event type for access available:

Example:
a) *Costs = 1.5d*
b) *ShipmentId = 123134*

The above example simply checks if the *Costs* equal the double type value *1.5d*. As there is only the *ShipmentCreated* type available it is not necessary to address its type explicitly. The comparison capabilities such as the equals operator will be discussed later on in this chapter.

6.2.1.2 Accessing Header Attributes

eventExprSpecial ⇒	*eventAtom AT IDENTIFIER* ∨
	AT IDENTIFIER ∨
	eventAtom LBRACK plusMinExpr RBRACK
	(DOT eventExpr)?

Event object types contain also header attributes to encapsulate special attributes for capturing metadata such as the creation time or priority of an event. To provide access to those header attributes there is the rule *eventExprSpecial* in place. The header attributes of event types can be accessed just like attributes, but instead of the *dot* the *at* symbol is used to separate the type and the attribute.

Example:
a) *ShipmentCreated@priority*

The example above shows the access to the header attribute *priority* of the event object type *ShipmentCreated*. In case that there is only one event object type available, it is not necessary to write *full-qualified attributes* just like with the normal attribute access.

6.2.1.3 Accessing Collections

Mentioned earlier in the introduction, the EAExpressions support multi-value types such as collections and dictionaries as attribute types. The following two rule expression paths are used to represent collection access. During the evaluation process of a given EAExpression the query processor checks if the accessed attribute is really a collection.

dotExpr ⇒	*eventAtom (DOT dotExpr)?* ∨
	eventExprSpecial ∨
	[...]
eventExpr ⇒	*eventAtom (DOT dotExpr)?* ∨
	[...]
eventExprSpecial ⇒	*[...]*
	eventAtom LBRACK plusMinExpr RBRACK
	[...]

The rule expression *eventAtom LBRACK plusMinExpr RBRACK* allows to place brackets after an attribute to access the index of an identifier.

Example:
a) *ShipmentCreated.Products[0].ProductName*

The above example accesses the collection attribute *Products* of the event object type *ShipmentCreated* at the index 0. The *Products* attribute is a nested type of the event object type *Product* that contains additional attributes such as a name, id and a price. Thus the *ProductName* is an attribute of the event object type *Product*.

6.2.1.4 Accessing Dictionaries

Dictionary attributes are lists of key-value pairs containing either a value represented as a runtime type or as an event object type. The key, which is the accessor for the dictionary value element, must be defined as a runtime type such as an integer or a string. From the syntax definition perspective, dictionaries are accesses through the same bracket enclosure notation like collections. The evaluation check, if the attribute is a valid dictionary and the key accessor is of the same type as the key type, is performed during the evaluation of the query.

Example:
a) *ShipmentCreated.ShipmentItems['55532']*

Lets consider the above example where we have an *ShipmentCreated* event object type containing an attribute *ShipmentItems*. In contrast to the previous collection example the *ShipmentItems* attribute is now a dictionary type, where the key is a string containing the container-id and the value containing the delivery date/time value. The above expression selects the *ShipmentItem* item with the key *'55532'* and returns the corresponding date/time value.

6.2.1.5 Expressing Event Object Aggregations

dotExpr ⇒	*eventAtom (DOT dotExpr)?* ∨
	eventExprSpecial ∨
	functionExpr
eventExpr ⇒	*eventAtom (DOT dotExpr)?* ∨
	eventExprSpecial
functionExpr ⇒	*functionAtom LPAREN parameterList RPAREN*
	(eventFunction)?
functionAtom ⇒	*IDENTIFIER*
parameterList ⇒	*boolExpr (COMMA boolExpr)**

According to the syntax definition there are two rules, the *dotExpr* and the *eventExpr*, which consist of almost the same rule expressions. The difference is that the *dotExpr* extends the rule body with the additional *functionExpr* path that allows to access so called event object aggregations and to define function constructs. In the latter case this it is omitted in certain syntax constellations such as after brackets in index accessors.

An event object aggregation comes to application if a query should access multiple instantiated event objects during runtime such as filtering out event object types with specific attributes. This can be the case inside correlations where there are several *ShipmentCreated* event objects that contain a self-correlation on their *ShipmentCreated.TransportInfo.Destination* attribute. Such a correlation collects all *ShipmentCreated* events during runtime that have the same transport destination.

Example:
a) *ShipmentCreated(Costs > 150).ProductName*
b) *ShipmentCreated(Costs > 150)@priority*

The above example shows who EAExpression can be used to formulate an expression that filters all *ShipmentCreated* events from a correlation where the *Costs* are greater than 150. The result of this expression would be a list of *ProductNames*. The aggregations can be seen as a kind of filter where it is possible to place inside the braces an arbitrary boolean resulting expression.

In order to be able to access the attributes of an aggregation result there is the additional rule **eventFunction** in place:

eventFunction ⇒	*DOT eventExpr* ∨
	AT IDENTIFIER ∨
	LBRACK plusMinExpr RBRACK (DOT eventExpr)?

This rule makes a distinction between three types of cases that can be applied after aggregation and function expressions.

- The first is a simple event object type attribute access with the dot notation.
- The second allows to access the header attribute of the result of an aggregation.
- The last one allows to access a specific index of a resulting event object collection followed by an optional attribute access.

The rule name *functionExpr* might be misleading in this context, but from a syntactical point of view aggregations and functions are ambiguous. Therefore they have the same syntax definition. The difference is made during evaluation phase of the abstract syntax tree by checking if the identifier left of the opening brace is an available event object type.

6.2.1.6 Functions

Mentioned in the previous section the difference of the syntactical definition of aggregations and functions are equal and the decision between functions or aggregations is made during the evaluation of the expression. A function expression consists of a *functionAtom*, which is the function name, a list of comma-separated parameters whereas each parameter allows the full expressiveness of EAExpressions and a return type as the result of a function.

Example:
a) *Avg(ShipmentCreated(ProductName = 'Tonsalumn').Costs))*

The above example shows a combination of an aggregation filter applied to the *ShipmentCreated* type and a function call that calculates the average value out of a given collection of values. In this example the *ShipmentCreated* events with the product name *Tonsalumn* are filtered (out of a correlation set for instance). The returned result of this filter is a collection of the costs which then is added as a parameter to the function *Avg*. This function takes one collection attribute with numeric values as a parameter and returns the average of that collection.

This is one of many functions offered by EAExpressions. They provide a large set of pre-implemented functions containing various mathematical, statistical and other common functions such as type converters or list operators.

6.2.1.7 Multi-Value Operators

collectionExpr ⇒	*LCURLY parameterList RCURLY*
specialKeywordExpr ⇒	*defExpr ((CONTAINS* ∨ *CONTAINSVALUE* ∨ *CONTAINSANY* ∨ *COLON) specialKeywordExpr)?*

EAExpressions ship, besides various collection related functions, also a set of native language operators for accessing multi-value operators. So for example there is the possibility to create collections out of attributes by placing curly brackets around expressions like shown in the following example:

Example:
a) *{'item1', 'item2'}*

The *COLON* operator allows to merge lists or values together. The first example shows the result of two merged lists and the second example merges three separate values. This operator allows to merge every type of values or objects that are supported by collection attributes.

Example:
a) *{1,2} : {3,4} : 5* → *{1,2,3,4,5}*
b) *1 : 2 : 5* → *{1,2,5}*
c) *ShipmentCreated : ShipmentCreated* → *{EventObject, EventObject}*

The *CONTAINS* operator checks if a collection contains a given value or a collection of values. If this operator is applied to a dictionary it checks whether the dictionaries key contains a given value or not.

Example:
a) *ShipmentCreated.ShipmentItems CONTAINS '234235'*
b) *ShipmentCreated.ShipmentItems CONTAINS {'234235', '234335'}*

The *CONTAINSVALUE* operator can be applied to dictionaries to check if a value is contained in the value part of a dictionary. In the following example the *ShipmentItems* value is checked if it contains the date '10-01-2008'.

Example:
a) *ShipmentCreated.ShipmentItems CONTAINSVALUE ToDateTime('10-01-2008')*

The *CONTAINSANY* operator can be applied both to collections and dictionaries. In contrast to the *CONTAINS* operator only one value of a list of values must be found. In the following example the list of *{1,2,3}* is checked if the values *2* or *5* occur.

Example:
a) *{1,2,3} CONTAINSANY {2,5}*

6.2.1.8 Boolean Operators

boolExpr ⇒	*conditionExpr ((AND ∨ OR ∨ XOR) boolExpr)?*
	∨ NOT boolExpr

The boolean expression provides three binary boolean operators *AND, OR and XOR* for evaluating combinations of additional expressions. Further it offers the unary *NOT* operator for applying negations against boolean expressions. With such boolean expression it is possible to create expression such as:

Example:
a) *expression AND expression OR expression NOT expression*

6.2.1.9 Comparison Operators

The definition of the next operator levels are shown below and include conditional expressions for comparing expressions. The *conditionExpr* defines the binding levels for the binary operators =, >, ≥, < and ≤.

conditionExpr ⇒	*isExpr ((EQUAL ∨ NOTEQUAL ∨ GTHAN ∨ GEQUALTHAN ∨ LTHAN ∨ LEQUALTHAN) conditionExpr)?*
isExpr ⇒	*plusMinExpr (IS_ OP typeExpr)?*

A level deeper and thus with a higher binding between expressions is the *IS_ OP* operator. This operator is intended to allow the comparison of evaluation results of subexpression. This is sometimes necessary in order to determine the result value of an operation.

Example:
a) *ShipmentCreated > 12*

6.2.1.10 Arithmetic Operators

plusMinExpr ⇒	*prodDivExpr ((PLUS ∨ MINUS) plusMinExpr)?*
prodDivExpr ⇒	*prodDivExprMod (STAR prodDivExpr)?*
prodDivExprMod ⇒	*prodDivExprDiv (MOD prodDivExprMod)?*
prodDivExprDiv ⇒	*specialKeywordExpr (DIV prodDivExprDiv)?*

The rules starting from *plusMinExpr* represent the arithmetic operations plus, minus, multiplication, division and modulo. Arithmetic operations are defined within a structure that nests *prodDivExprDiv* ⇒ *prodDivExprMod* ⇒ *prodDiv-*

Expr ⇒ *plusMinExpr* into each other. This is done to preserve the binding nature of operators such as a multiplication has a higher priority for evaluation than an addition.

Example:
a) *expression + expression - expression / expression*
b) *expression % expression * expression / expression*
c) *expression = expression AND expression ≥ expression ≥ 5*

The EAExpression syntax preserves the binding strength through the syntax level definition. The farther the rules go down the stronger their bind becomes. By applying parentheses, the binding can rearranged just like in common mathematics. Expressions enclosed by parentheses are evaluated before the other operators are applied.

defExpr ⇒	*constValue* ∨
	dotExpr ∨
	collectionExpr (LBRACK plusMinExpr RBRACK)? ∨
	LPAREN boolExpr RPAREN ∨
	MINUS defExpr ∨
	PLUS defExpr ∨
constValue ⇒	*STRING_LITERAL* ∨
	CONSTANT ∨
	TRUE ∨
	FALSE ∨
	constNumericValue

Further, EAExpressions allow to define constants that can be string literals or some defined identifier. They can be defined while setting up the EAExpression query processing instances. From the start away there are the two constants TRUE and FALSE as an equivalent for the logical true and false in boolean environments.

6.2.2 EAExpression Syntax Tree Model

The ANTLR parser uses the EAExpression syntax definition rules, introduced in the previous section, to build up an abstract syntax tree in memory for further validation and evaluation during runtime. This section introduces the EAExpression abstract syntax tree for the language constructs that can be expressed. For validating the syntax, the tree is traversed through for checking its semantic correctness. So for instance, if a dot notation is used to access the attributes of an event object type, the query engine has to check, besides its syntactical correctness, if the event object type and the accessed attribute exists. Further validations like compatible types in comparisons or in arithmetic operations have to be taken into consideration.

ANTLR supports a declarative prefix notation for describing the structure of abstract syntax trees (AST). The syntax tree is built up based on the EBNF rules. Each non-terminal in a rule causes an expansion of a subtree. However ANTLR extends the EBNF with several special functionalities that allow to enrich the nodes with additional metadata or to ignore tree expansions.

Figure 6.2 shows an illustration of a syntax tree that represents the simple arithmetic expression: *1 + 2 + 3*. A tree parser rule that describes the token constellations would like as follows:

arithmetic $\Rightarrow$	PLUS arithmetic arithmetic $\vee$
	MINUS arithmetic arithmetic $\vee$
	NUMERIC

The syntax trees are described with a prefix notation whereas the outermost left token is a root and the rest of the tokens are leafs. Each sub rule, like the recursive call of the *arithmetic* rule in this example, results in one step deeper of the tree. The termination (i.e. the stop rule) is a terminal, like in this case the *NUMERIC* symbol. Basically the rules define the tree patterns and their meanings. ANTLR uses those patterns to trigger the evaluation of specific operations to evaluate the related nodes of an AST. If the *arithmetic* rule matches, a function can be called to add the two parameters together. If there is still a node left, it is recursively called and the result is put on the stack and worked down later.

In the following sections the event object type example presented in Figure 6.1 will be reused for explanatory purposes.

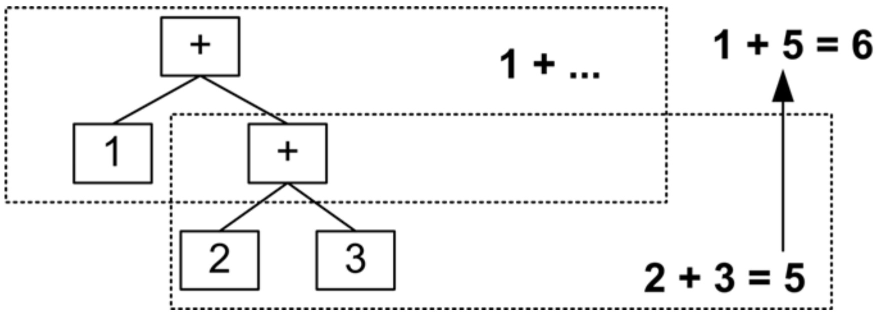


Figure 6.2: Example of a simple arithmetic tree

6.2.2.1 Event Object Type AST Representation

There are basically two types of event object type accesses. The first one is the access of attribute in a *full qualified attribute* manner - that is by providing the event object type identifier followed by a dot. The abstract syntax tree for the following expressions are illustrated in Figure 6.3.

Example:
a) *ShipmentCreated.Costs*
b) *ShipmentCreated.TransportInfo.Destination*

The AST parser rules are defined as follows. The root node is always a dot symbol from where subexpressions (*dotExpr*) are hung on to the left and the right side. Due to the recursive definition an arbitrary deep tree can be described whereas the right leaf would be another dot root for two expressions.

dotExpr ⇒	IDENTIFIER ∨
	DOT dotExpr dotExpr ∨
	eventExprSpecial ∨
	functionExpr ∨

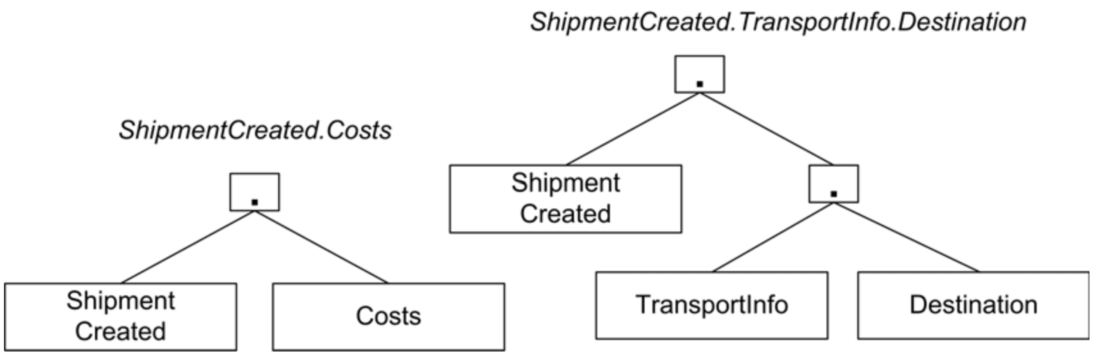


Figure 6.3: Event Object Type AST Representation

6.2.2.2 Header Access AST Representation

The header attributes are represented as a binary tree just like the normal dot notation described in the previous section. Further there is a *full qualified attribute* notation where the event object type is mandatory.

eventExprSpecial ⇒	*AT dotExpr IDENTIFIER* ∨
	AT IDENTIFIER ∨
	LBRACK defExpr plusMinExpr

The representation of the following example is illustrated as an abstract syntax tree in Figure 6.4.

Example:
a) *ShipmentCreated@priority*

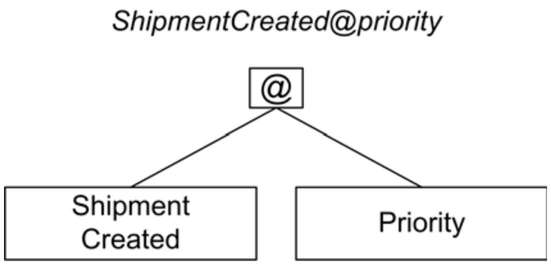


Figure 6.4: Header Access AST Representation

6.2.2.3 Collection AST Representation

Collections are represented in the abstract syntax tree truncated as not every symbol is required to be represented for validation and evaluation. The previously introduced rule *eventExprSpecial* with the body rule *LBRACK defExpr plusMinExpr* represents the tree pattern for collections. For identifying an index accessor it is enough to have one opening bracket as the root node in place.

Example:
a) *ShipmentCreated.Products[12].ProductID*

The tree representation of the above example is illustrated in Figure 6.5. In this example the collection type attribute of the event object type *ShipmentCreated* is accessed at the index 12. To provide a better view how these AST rules play together the dot notation was combined.

Custom created collections are defined through the meta annotation *COLLECTION* in a node and contain a parameter list annotated as *PARAMETER_LIST* while they have an arbitrary number of children containing the list elements. Figure 6.6 shows this special annotation tree construct resulting from the parser execution.

collectionExpr ⇒	*COLLECTION parameterList*
parameterList ⇒	*PARAMETER_LIST (boolExpr)**

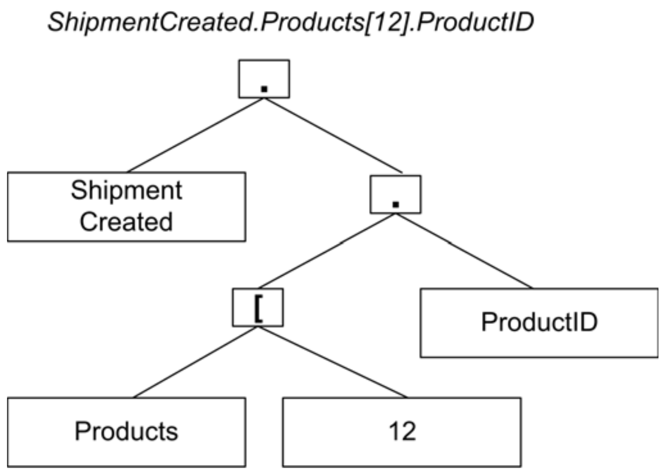


Figure 6.5: Collection Access AST Representation

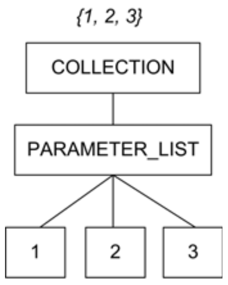


Figure 6.6: Custom Created Collection AST Representation

6.2.2.4 Dictionary AST Representation

The representation of the dictionary AST is similar to the collections tree representation. The syntax definition itself does not restrict the content enclosed by brackets to numeric values. Therefore from the syntactical and the AST point of view there is no difference between accessing collections and dictionaries. However, during the evaluation of the tree, the attribute type is checked and in case of a syntax error an exception with a detailed problem description is thrown.

6.2.2.5 Event Object Aggregation Filter AST Representation

Event Object Aggregations are treated as functions from a syntactical and abstract syntax tree point of view. The aggregation has the semantic meaning of a filter that allows to apply filtering conditions on attributes of sets of event objects. During syntax parsing the ANTLR rules enrich the abstract syntax tree with additional meta data in order to be able to make a distinction between several special cases of function creation.

functionExpr ⇒	*FUNCT IDENTIFIER parameterList (functionExpEval)?*
functionExpEval ⇒	*FUNCTION_EVENT_EVAL dotExpr* ∨ *FUNCTION_EVENT_HEADER_EVAL headAttr1* ∨ *FUNCTION_COLLECTION_EVAL plusMinExpr (plusMinExpr)?*

The simplest form of a resulting aggregation is a node with children containing the parameter expressions. A function node is flagged with a *FUNCT* label. In case that the attribute of a resulting aggregation collection is accessed the node is flagged as *FUNCTION_EVENT_EVAL*, if the header attributes are accessed, the node is flagged as *FUNCTION_EVENT_EVAL* and if the index is accessed it is labeled as *FUNCTION_COLLECTION_EVAL*. For better understanding lets consider the following example:

Example:
a) *ShipmentCreated(Costs > 5).Products.ProductID*

The Figure 6.7 illustrates the corresponding abstract syntax tree to the above stated example. The *PARAMETER_LIST* node contains the filter expressions inside the braces whereas the *FUNCTION_EVENT_EVAL* holds accessors after the the closing braces as children.

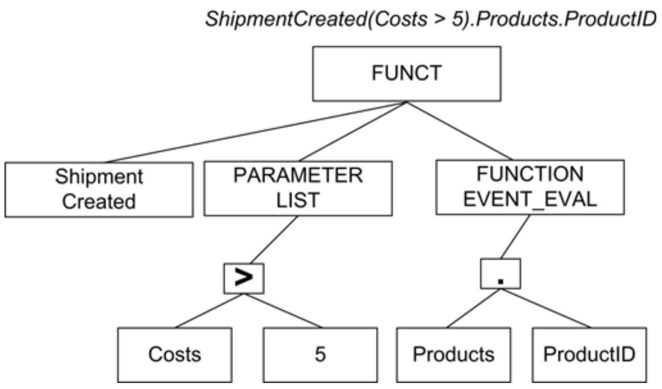


Figure 6.7: Aggregation Access AST Representation

6.2.2.6 Function AST Representation

The abstract syntax tree definition for the function is exactly the same as for aggregations, but with a different semantic meaning. So for instance the tree is allowed to hold more than one parameter according to the functions definition. The validity of the function signature, such as type correctness, is represented by the tree and checked during the evaluation processes. The EAExpression functions can be extended by inheriting new functions from an abstract base class and then implementing the methods according to the functions requirements.

Example:
a) *If(ShipmentCreated.Costs >= 50.0, 'Expensive', 'Cheap')*

Lets consider the above example making use of functions. The Figure 6.8 illustrates the corresponding AST representation. The *If*-function take three arguments where the first one requires a condition, evaluating to a boolean result, the second parameter is the result on if the condition evaluates to true and the last one is the result on a false evaluation. The *PARAMETER_LIST* holds three nodes, each one for the parameters separated by a comma.

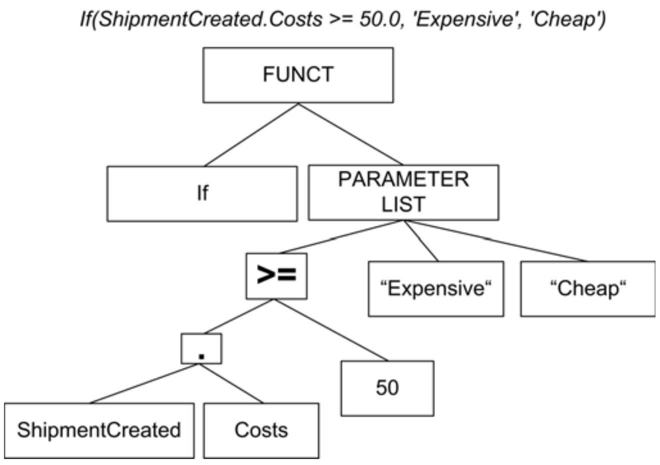


Figure 6.8: Function AST Representation

6.2.2.7 Boolean Expression AST Representation

The boolean operators representation is a straight forward binary tree where the root is the operator and the two child nodes are the expressions on which the operators are applied. The only exception is the *NOT* operator which only has one child element.

boolExpr ⇒ *AND boolExpr boolExpr* ∨
OR boolExpr boolExpr ∨
XOR boolExpr boolExpr ∨
NOT boolExpr ∨
conditionExpr

Figure 6.9 shows the tree representation of the example stated below.

Example:
a) *true AND true OR NOT false*

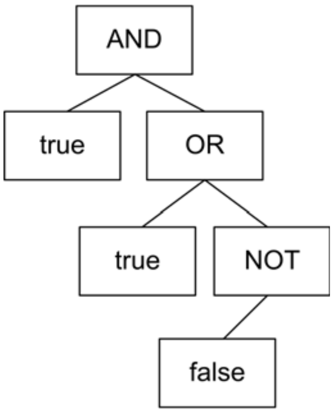


Figure 6.9: Boolean Expression AST Representation

6.2.2.8 Comparison Expression AST Representation

The comparison operator representation is also a binary tree where the root is the operator and the two child nodes are the expressions on which the operators are applied.

conditionExpr $\Rightarrow$	*EQUAL boolExpr boolExpr* $\vee$
	NOT_EQUAL boolExpr boolExpr $\vee$
	LTHAN boolExpr boolExpr $\vee$
	LEQUALTHAN boolExpr boolExpr $\vee$
	GTHAN boolExpr boolExpr $\vee$
	GEQUALTHAN boolExpr boolExpr $\vee$
	isExpr
isExpr $\Rightarrow$	*IS_OP boolExpr typeExpr* $\vee$
	plusMinExpr

Figure 6.10 shows the tree representation of the example stated below.

Example:
a) $3 > 2 > 1$

6.2.2.9 Arithmetic Expression AST Representation

The arithmetic operators are represented in a binary tree style whereas the operator is the root node for the child expressions.

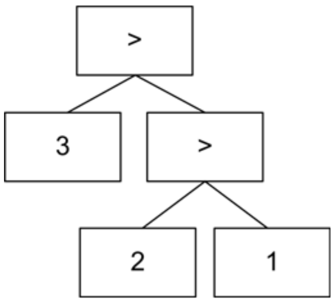


Figure 6.10: Comparison Expression AST Representation

plusMinExpr $\Rightarrow$	*PLUS boolExpr typeExpr* $\vee$
	MINUS prodDivExpr $\vee$
	prodDivExpr
prodDivExpr $\Rightarrow$	*STAR boolExpr typeExpr* $\vee$
	DIV boolExpr typeExpr $\vee$
	MODE boolExpr typeExpr $\vee$
	specialKeywordExpr

Figure 6.11 shows the tree representation of the example stated below. This example also highlights the operator binding strengths as well as the *overload* of the binding by enclosing expressions with braces.

Example:
a) *(1 + 2) * 4 / 5*

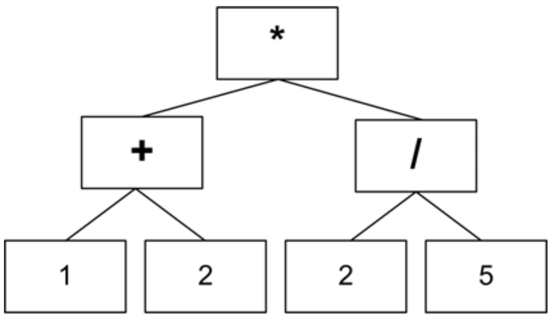


Figure 6.11: Arithmetic Expression AST Representation

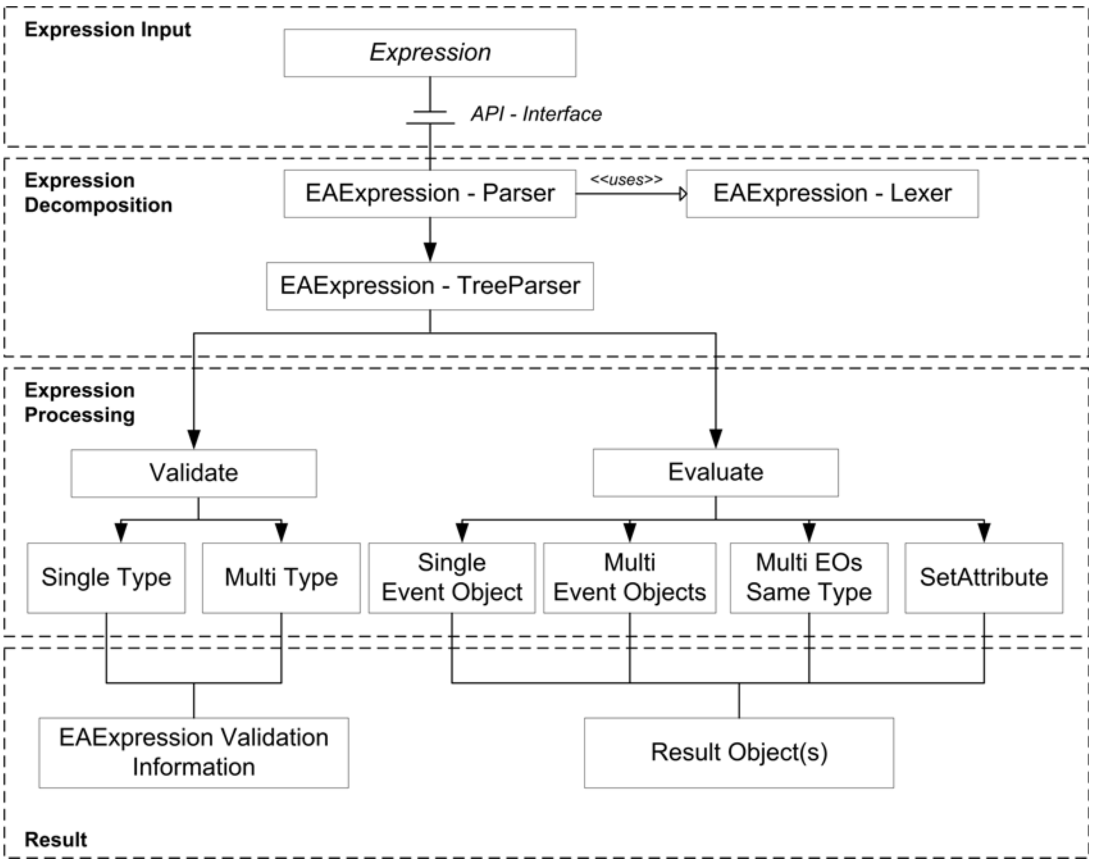


Figure 6.12: EAExpression Processing Overview

6.2.3 EAExpression Evaluation Model

The EAExpression evaluation process is described from a high-level perspective and illustrated in a simplified form in Figure 6.12. The expression processing procedures can be split up into four layers. Due to the extremely high implementation complexity of the EAExpressions this section only introduces an overview of the validation and evaluation processes of EAExpressions. EAExpressions are a domain specific subset language in context of the SARI-SQL query languages. As the main focus of this dissertation is the query language itself, EAExpressions can be treated from a validation and evaluation point of view as a black-box. The important concepts of EAExpressions are the syntax and semantic definitions and furthermore the abstract syntax tree representation as some concepts are overridden by SARI-SQL in order to be able to define constraints. However, this section provides the basic overview of the black-box validation and evaluation process of EAExpressions for the sake of completeness.

6.2.3.1 Expression Input

The *Expression Input* process layer provides an API interfaces towards the processing engine and the result delivery. EAExpressions can be triggered for evaluation or validation through this API. The main building blocks of the *Expression Processing* layer also defines the signature variations of the API input. So for instance it is possible to validate an EAExpression against several event object types or against only one. The latter case does not require a full qualified notation (compare with Section 6.2.1.1).

6.2.3.2 Expression Decomposition

The *Expression Decomposition* layer is responsible for parsing the EAExpressions according to predefined rules and transforming it into an abstract syntax tree, leaving away unnecessary symbols for processing and enriching nodes with additional meta-data. The abstract syntax tree parser is then in charged of traversing through the tree elements and depending on the type of the expression processing either validating or evaluating the given expression.

The EAExpression decomposition consists of three main components. A lexer (lexical analyzer, string tokenizer), a parser and an abstract syntax tree parser that walks through the syntax tree.

The lexer is the component that scans a text stream character by character and creates token quantifications of recognized character sets. A token is a defined group of text. Characters that have no semantic meaning for the language such as white spaces or comments are removed. Recognized character sets (i.e. tokens) can than be used later on in the syntax to define rules for parsing an expression or in general the language. While the characters are tokenized they receive a meaning. Such tokens are for operators, keywords or identifiers. For instance an *IDENTIFIER* is defined as a set of coherent literal characters. Errors are generated if the lexer detects sequences of character that do not match the lexical rules.

6.2.3.3 Expression Processing

The *Expression Processing* process layer is separated into two major processing types. The first is the validation, which basically checks the semantics of an expression against event object types to ensure the typing correctness for instance. The second is the evaluation, which is applied during runtime against instantiated event object. The class hierarchy of the access points is shown in figure 6.13

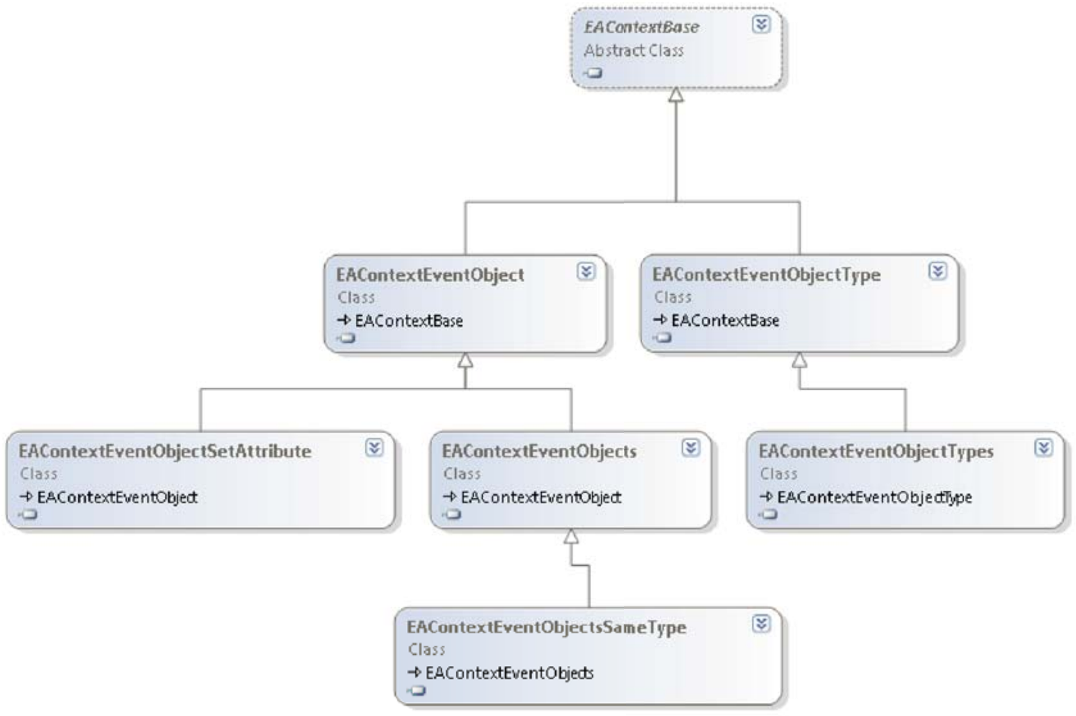


Figure 6.13: EAExpression Tree Parser Class Diagram

Finer granularity levels in the options of validating and evaluating events are available for validation as well as evaluation of EAExpressions. The validation can be performed with the following two options:

- **Single Type (EAContextEventObjectType):** This option validates an EA-Expression against a single event object type. It allows to express EAExpressions without addressing event object types in a full qualified notation (compare with Section 6.2.1.1).

- **Multi Type (EAContextEventObjectTypes):** This option validates an EA-Expression against several event object types. This means that EAExpressions have to be expressed in a full qualified notation and thus require that an event object type identifier has to be placed before the accessed attribute separated by a dot.

The evaluation against event object instances during runtime can be performed through the following four evaluations options:

- **Single Event Object (EAContextEventObject):** This evaluation option allows to evaluate exactly one event object instance against an EAExpression. The corresponding event object type is resolved from the given event object instance (i.e. retrieved from the type library). This option is mainly used for debugging and testing purposes.
- **Multi Event Object (EAContextEventObjects):** This evaluation options is used to evaluate several event object instances against an EAExpressions. It is applied whenever a set of events with different event object types must be evaluated on given expression. This option resolves the event object types by itself as there is a full qualified notation in place which enforces the use of the corresponding event object types.
- **Multi EOs Same Type (EAContextEventObjectsSameType):** This option evaluates a set of event object instances against one specific event object type that must be provided. It allows to apply an EAExpression against several event objects that have the same type and thus don't need full qualified notations.
- **SetAttribute (EAContextEventObjectSetAttribute):** This option is a specialty that allows to run an EAExpression for evaluation against one event object in order to set an attribute to a different value. This evaluation option is used to modify event attributes in the event processing maps of SARI (compare with Section 5.1.3).

6.2.3.4 Expression Result

The result of an EAExpression validation process is a detailed listing of syntactical or semantical errors in case that an error occurred. The evaluation results are a list ob objects of a specific type have been retrieved through an EAExpression.

6.3 SARI-SQL

6.3.1 SARI-SQL Syntax Definition

In the preceding sections the capabilities and features, that can be expressed by the SARI-SQLs syntax, will be introduced and discussed in detail. The SARI-SQL syntax structure is basically relying on four main key clause-constructs:

- ***SELECT.*** The *SELECT* clause represents the projection part of the query like known from ANSI-SQL. It allows to select attributes, but in contrast to normal SQL these are attributes of event object types instead of tables (i.e. relations). Furthermore, each of the projections in this clause allow to define EAExpressions with certain restrictions that will be explained later on in this chapter.

- ***FROM.*** The *FROM* clause defines the data sources, that can be either an event object type, a metric or a score information, that are under examination in terms of applicable in the projection (*SELECT*) clause and the condition (*WHERE*) clause. In the relational database SQL world the *FROM* clause allows to select relations that consist of attributes (e.g. columns). In SARI-SQL these are the attributes of event object types or metrics and scores of a specific type.

- ***WHERE.*** The *WHERE* clause is the conditional part of an SARI-SQL query and allows to define filters upon attributes and further it allows to create inner joins over event object type attributes similar to the relational model.

- ***OVERCORR.*** The *OVERCORR* construct allows to define a kind of a preselector over events by restricting the space of available event object types. The space restriction is determined by the selected correlation in the clause.

The preceding section will explain in detail the syntax of SARI-SQL and itŠs capabilities step-by-step. Like with EAExpressions in the previous sections, there is the full syntax definition and a complete syntax diagram collection available in the appendix in Section 9.3. SARI-SQLs syntactical capabilities and the semantics will be explained with the help of examples. In contrast to EAExpressions the concepts of the SARI-SQL language is much more compact because a lot of functionalities are encapsulated inside the EAExpressions that are nested within SARI-SQL constructs. The complexity of SARI-SQL is lying under the hood within the evaluation process of a given query. Therefore, special attention will be paid to the evaluation and optimization processes.

Figure 6.14 shows a compilation of three event object types (*ShipmentCreated, TransportStart, TransportEnd*) forming the two correlations *ShipmentToTransport* and *TransportInfo*. This example will be used to explain the syntactical concepts of SARI-SQL throughout in this chapter.

The following syntactical definition shows the main entry point for the SARI-SQL syntax. The *selectExpression* defines the four main clauses whereas only the *SELECT* (e.g. the projection) and the *FROM* clause is required. Selecting events from correlations or applying filters or creating joins is not mandatory just like in common ANSI-SQL.

sariSQL ⇒	*(selectExpression)*
selectExpression ⇒	*((SELECT selectClause) (FROM fromConstruct)*
	(OVERCORR overCorrClause)?
	(WHERE whereCondition)?)
	(ORDER_BY orderByCondition)?)

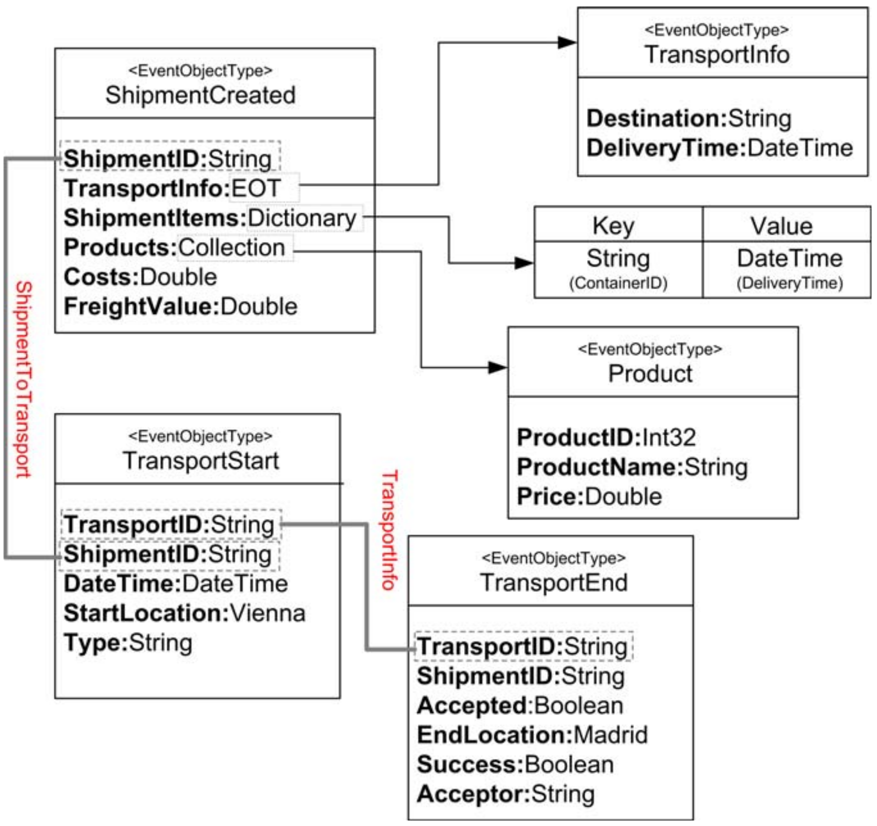


Figure 6.14: Event Object Type Example for the SARI-SQL

6.3.1.1 Retrieving Event Object Types, Scores and Metrics

fromConstruct	$\Rightarrow$	*fromList* $\vee$ *joinClause*
fromList	$\Rightarrow$	*fromItem (COMMA fromItem)**
fromItem	$\Rightarrow$	*reducedDotExpr (fromAlias)?* $\vee$ *functionExpr (fromAlias)?*
reducedDotExpr	$\Rightarrow$	*eventAtom (DOT reducedDotExpr)?*

The *FROM* clause is the construct that allows to define the accessible event object types, their attributes, metrics and scores. The clause takes a list of arguments defined in the *fromList* rule and allows to create various join constructs defined by the rule *joinClause*. The join constructs are excluded in this section and will be explained later on in detail in the proceeding sections. There are two alternatives for defining the *fromItem*. The first one defines a list of comma separated items that represent the identifier of an event object type. The second one takes a *functionExpr* a comma separated list of function like expressions that define the identifier of metrics and scores. The *fromAlias* item is an optional identifier for defining alias names for event object types and metrics/scores. This aliasing is comparable to the table aliasing in SQL. In SARI-SQL aliasing is mandatory if more than one event object type or scores/metrics are in the from clause. The *fromItem* rules take a *reducedDotExpr* for allowing to create a concatenation of dots with identifiers. That is required in order to place and alias accessor of correlations before an event object type. This feature is covered and discussed later on with the *OVERCORR* clause in Section 6.3.1.6.

Example:
a) *SELECT * FROM TransportStart*
b) *SELECT start.TransportID FROM TransportStart start, TransportEnd end*
c) *SELECT * FROM Metric('AvgTransportDuration')*

The first query of the above examples selects all in the Event-Base available events of the type *TransportStart*. The second example selects all available events of the types *TransportStart* and *TransportEnd*. In addition the two types are aliased with the labels *start* and *stop*. Aliasing is used to handle the ambiguity of items defined in the *FROM*. For instance both types *TransportStart* and *TransportEnd* contain an attribute that is called the same (*TransportID*). In this example the *TransportID* of the *TransportStart* event object type in the *SELECT* clause is retrieved. The last example selects all classifiers from the metric *AvgTransportDuration*. Each classifier of the metric is treaded as a column in the *SELECT* clause and thus it is also possible to define joins or constraints in the *WHERE* clause on them.

6.3.1.2 Projecting Event Object Type Attributes

selectClause	⇒	*selectList*
selectList	⇒	*((selectItem) (COMMA selectList)?)*
selectItem	⇒	*plusMinExpr (AS displayName)?*
displayName	⇒	*STRING_LITERAL* ∨ *IDENTIFIER*

The *SELECT* clause consists of a list, defined by the rule *selectList*, of one or more *selectItem*s that are separated by a comma. Each of the list items represent one projection clause that is materialized as a column in the resulting table similar to SQL. A *selectItem* is defined by the syntax of a *plusMinExpr* which is a rule that is part of the EAExpression syntax.

The SARI-SQL syntax definition is derived from the EAExpression syntax and therefore every rule definition from it is reusable. The idea behind placing EAExpressions into the projection clause is to pay a special attention to the nature of the event typing concepts of SARI. The reader should bear in mind that, in contrast to SQL, there are event objects, respectively event object types, that are under examination with their special characteristics. Those special event related characteristics were discussed in detail in Section 5.2 like the attribute structures. In SARI-SQL however, the EAExpression in projections clauses exclude the ability to express boolean and conditional expressions as they both result to a boolean value. Therefore the starting rule for projections is set to *plusMinExpr* and not to a higher level rule of EAExpressions such as *conditionExpr* or *boolExpr*.

Example:

a) *SELECT TransportID, StartLocation FROM TransportStart*
b) *SELECT start.StartLocation, end.EndLocation*
FROM TransportStart start, TransportEnd end
c) *SELECT ShipmentID, FreightValue - Costs, Avg(Product.Price)*
FROM ShipmentCreated

The first of the above examples returns a list consisting of two attributes (i.e. columns) whereas the columns are the attributes *TransportID* and *StartLocation* of the event object type *TransportStart*. The second example selects two different attributes (*StartLocation* and *EndLocation*) from two different event object types (*TransportStart*, *TransportEnd*). As there is more than one event object type in the *FROM* clause it is required to define an alias for them. As aliasing is activated it is also mandatory to apply it in every other clause in order to be able to access the attributes of event object types and avoid ambiguity. This is an important point, as in EAExpressions, the dot notation separates event object type identifiers from the attributes. In SARI-SQL the outer most left identifier in a dot concatenated expression is an alias if there is more than one event object type in the *FROM* clause. The last example illustrates the possibility of applying EAExpressions in the *SELECT* clause items such as the *Avg*erage function on the collection of *Prices*.

The *SELECT* clause also allows to apply the star (*) operator to select items. The star operator is comparable to the star in the *SELECT* clause of standard SQL. It selects every attribute of the defined event object types in the *FROM* clause. By applying aliases to the star, it is possible to define on which event object type the star should be applied. In case that there is more than one item in the *FROM* clause and one star operator is in the *SELECT* clause, then every attribute of every available event object type will be selected for projection.

However there is one major specialty with the star operator in SARI-SQL. The nature of event object types describes that there are so called multi-value attributes types where dictionaries, collections and nested event object types as attributes belong to. As these type of attributes represent multidimensional data they are returned as data objects preserving the data dimensions. This is an important difference to the relational model where the structures (i.e. relations) are "flat".

Example:

a) *SELECT * FROM TransportStart*

b) *SELECT start.*, end.**
FROM TransportStart start, TransportEnd end

The above example shows two examples of the star syntax usage in the *SELECT* clause.

6.3.1.3 Defining Conditions

whereCondition ⇒	*boolExpr*
boolExpr ⇒	*LPAREN selectExpression RPAREN* ∨ *conditionExpr ((AND* ∨ *OR* ∨ *XOR) boolExpr)?* ∨ *NOT boolExpr*
textit**conditionExpr** ⇒	*isExpr ((EQUAL* ∨ *NOTEQUAL* ∨ *GTHAN* ∨ *GEQUALTHAN* ∨ *LTHAN* ∨ *LEQUALTHAN) conditionExpr)?*

The *WHERE* clause is similar to SQL which represents a condition that can be applied to a set of tuples to reduce the results or to perform joins over sets of tuples. However in SARI-SQL the tuples are defined by event object types and their attributes and not by relations. The *WHERE* clause is defined by the *boolExpr* which overrides the *boolExpr* of EAExpressions. The original EAExpression *boolExpr* is defined as:

boolExpr ⇒	*conditionExpr ((AND* ∨ *OR* ∨ *XOR) boolExpr)?* ∨ *NOT boolExpr*

The main difference is that the overridden SARI-SQL *boolExpr* extends the original syntax by allowing to define sub-selects in *WHERE* clauses similar to SQL. In general every *WHERE* clause must result in a boolean return value which means that either a comparator or boolean operator must be applied. Defining a *WHERE* clause that contains only an arithmetic operation is omitted. Inner joins over attributes of event object types can be defined, similar to standard SQL, by applying an equals operator on the attributes of types that should be joined. Every EAExpression feature and function is fully available in *WHERE* clauses.

Example:
a) *SELECT * FROM ShipmentCreated*
WHERE Costs > 20 AND FreightValue < 100 AND
TransportInfo.Destination = 'Vienna'

The above example selects all *ShipmentCreated* events that are designated to be delivered to Vienna and where the shipment costs are greater than 20,– while the value of the freight is smaller than 100,–. This simple example illustrates the application of the *WHERE* clause with EAExpressions and exposes the ease of use of event object types specific characteristics.

6.3.1.4 Defining Implicit Time Windows

Event processing solutions, such as Esper [30], Aurora [1][92] or SASE [48], make use of continuous query languages for the purpose of creating queries over streams of events. For performance reasons these type of solutions provide means to define a time frame in which the events are valid to be ran against a query. This could be for instance a query that is calculating the average amount of attributes in a time frame of 15 minutes.

Events represent usually real world happenings and often information about the date and time is explicitly available as attributes in events. For instance a *ShipmentCreated* or a *TransportStart* event contain a *DateTime* attribute that represents when such a shipment has been created or when a transport went out.

Example:
a) *SELECT * FROM TransportStart*
WHERE 01.02.2008 < DateTime < 29.02.2008

The above example selects all *TransportStart* events that were conducted in February. This is the simplest way to create a time frame over a selection of events. However often information about timing is not explicitly modeled. This is were header attributes come into play (compare with section 5.2). Every event object in SARI contains event header attributes which are inherited from the *BaseEvent* type and capture metadata such as the creation time or priority of an event. The header attribute *timeCreated* contains the timestamp of the event creation.

Example:
a) *SELECT * FROM TransportStart*
WHERE 01.02.2008 < @timeCreated < 29.02.2008

This example shows how the *timeCreated* header attribute can be used to set a time frame over events independent from explicitly modeled timing related attributes.

6.3.1.5 Defining Joins

SARI-SQL supports five ways of defining join constructs and four different types of join operations. Just like in ANSI-SQL, SARI-SQL makes a distinction between inner joins and left/right/full outer joins with the same relational behavior.

joinClause ⇒	*fromSpecForJoin INNER JOIN fromSpecForJoin*
	ON onJoinClause ∨
	fromSpecForJoin FULL OUTER JOIN fromSpecForJoin
	ON onJoinClause ∨
	fromSpecForJoin LEFT OUTER JOIN fromSpecForJoin
	ON onJoinClause ∨
	fromSpecForJoin RIGHT OUTER JOIN fromSpecForJoin
	ON onJoinClause
fromSpecForJoin ⇒	*(fromSpec fromAlias)*
fromSpec ⇒	*IDENTIFIER (IDENTIFIER)?*
fromAlias ⇒	*IDENTIFIER*
onJoinClause ⇒	*onJoinItem*
onJoinItem ⇒	*dotExpr EQUAL dotExpr ((AND* ∨ *OR) onJoinItem)?*

A join can be defined always in the *WHERE* clause of a SARI-SQL statement simply by placing an equals operator between attributes of event object types. The following example illustrates a join between *TransportStart* and *TransportEnd* events:

Example:

a) *SELECT start.* FROM TransportStart start, TransportEnd end*
WHERE start.TransportId = end.TransportId

Inner joins, left-, right- and full-outer joins can be defined similar to SQL by placing the corresponding join type into the *FROM* clause. Additionaly the *ON* keyword defines which join conditions should be applied. The *ON* part of the clause is restricted by the rule *onJoinItem* to EAExpression dot notations that can be combined with the operators *OR, AND* or EQUAL. The following list of examples illustrates the usage of the join constructs:

Example:

a) *SELECT start.**
FROM TransportStart start INNER JOIN TransportEnd end
ON start.TransportId = end.TransportId

b) *SELECT start.**
FROM TransportStart start RIGHT OUTER JOIN TransportEnd end
ON start.TransportId = end.TransportId

c) *SELECT start.**
FROM TransportStart start LEFT OUTER JOIN TransportEnd end
ON start.TransportId = end.TransportId

d) *SELECT start.**
FROM TransportStart start FULL OUTER JOIN TransportEnd end
ON start.TransportId = end.TransportId

6.3.1.6 Retrieving Event Object Types of Correlations

Mentioned earlier in the Section 5.3, the concept of correlation, defining and managing the relationships of events, is a key characteristic of event processing solutions. A correlation defines a collection of semantic rules to describe how specific events are related to eachother. Such correlated events can be used to track causal relationships to calculate metrics. The internal representation and organization of correlations in the Event-Base is described in detail in Section 5.5 and is the underlying data structure for SARI-SQL.

From a SARI-SQL syntactical point of view correlations are described through a unique identifier such as *ShipmentToTransport* or *TransportInfo* like shown in Figure 6.14. The semantics of correlations in SARI-SQL is that they basically reduce the space of selectable event object types to the ones that have been gathered according to the correlation definition (compare with Section 5.3). That means that in case of a *TransportInfo* correlation only the *TransportStart* and *TransportEnd* event object types are selectable in the *FROM* clause and further, the set of event objects in the result, are restricted to correlated event objects that correspond to the selected correlation.

overCorrClause	⇒	*corrList*
corrList	⇒	*corrItem (COMMA corrItem)**
corrItem	⇒	*corrSpec (corrAlias)?*

In terms of the syntax the definition of correlation elements in SARI-SQL is quite simple. After the keyword *OVERCORR* the correlation name must be set. In case that the user wants to apply more than one correlation set, aliasing has to be applied, just like in the *FROM* clause. Otherwise the aliasing is optional. In case that no aliasing is provided and therefore only one correlation set is selected, all event object types in the *FROM* clause refer to the given correlation set. However if aliasing is activated the correlational aliases must be applied to the event object types in the *FROM* clauses as the first part of the dot notation. If a correlation alias is not applied to an event object type then the type is not retrieved from the correlation set and every event of that type is taken into account. This might be a handy feature when trying to create joins between already correlated events and events that have, from a correlation definition point of view, no relationship. From a certain standpoint the *OVERCORR* can be seen as a *FROM* clause that is at a higher hierarchy level.

Example:

a) *SELECT start.StartLocation, end.EndLocation*
FROM TransportStart start, TransportEnd end
OVERCORR TransportInfo WHERE start.StartLocation = 'Vienna'

b) *SELECT start.StartLocation, end.EndLocation*
FROM Corr1.TransportStart start, Corr2.TransportEnd end
OVERCORR TransportInfo Corr1, TransportInfo Corr2

The first example selects the *StartLocation* and *EndLocation* from the types *TransportStart* and *TransportEnd*. Both event object types must be contained in sessions of a *TransportInfo* correlation, with the restriction that the *StartLocation* of a transport is settled in Vienna. The second example takes two correlation sets which results in a cartesian product because of the two event object types in the *FROM* clause. This example is meant for illustrating the usage of aliasing with more than one correlation set used in a query.

6.3.1.7 Additional Constructs

orderByCondition	⇒	*orderByList*
orderByList	⇒	*orderByItem (COMMA orderByItem)**
orderByItem	⇒	*reducedDotExpr (ASC ∨C DESC)?*
groupByCondition	⇒	*groupByList*
groupByList	⇒	*groupByItem (COMMA groupByItem)**
groupByItem	⇒	*reducedDotExpr*
havingCondition	⇒	*boolExpr*

SARI-SQL supports also additional query constructs such as *ORDER BY*, *GROUP BY* or *HAVING* whereas the *HAVING* clause is granting EAExpressions as conditions and thus allows to apply their rich function set. From a syntactical point of view both *ORDER BY* and *GROUP BY* have almost similar syntactical rules whereas the condition of *HAVING* is represented through the *boolExpr* rule of the EAExpressions. From their semantical behavior these constructs are the same as in ANSI-SQL with the difference that event object types and their attributes are applied. Therefore additional information about the behavior of these functions can be found in contemporary literature about SQL.

Nevertheless here are a couple of examples about the usage:

Example:

a) *SELECT TransportID, StartLocation, Type*
FROM TransportStart
ORDER BY StartLocation, Type

b) *SELECT TransportID, StartLocation*
FROM TransportStart
GROUP BY StartLocation

c) *SELECT ShipmentID, TransportInfo.Destination, Sum(Costs)*
FROM ShipmentCreated
GROUP BY TransportInfo.Destination
HAVING Sum(Costs) > 100

6.3.2 SARI-SQL Syntax Tree Model

Just like with the EAExpressions, the ANTLR parser uses the syntax definition rules to build an abstract syntax tree in memory for validating and evaluating a given query. The goal of this section is to introduce the abstract syntax tree for the query constructs of SARI-SQL. Information about the nature of the notation, examples and the general structure can be found in Section 6.2.2. Basically SARI-SQL uses the AST representation of a query for traversing through the elements and creating an internal model of the query constructs for optimization and rewriting purposes. So actually, in SARI-SQL, only the AST representation itself is not sufficient. Topics such as evaluation and optimization processes will be discussed later on in Section 6.3.3.

In the following sections the event object type example presented in Figure 6.14 will be reused for explanatory purposes.

6.3.2.1 Top-Level Construct AST Representation

sariSQL ⇒	*QUERY selectClause fromConstruct (overCorrClause)?* *(whereCondition)?* *(orderByCondition)?* *(groupByCondition)?* *(havingCondition)?*

Every SARI-SQL query has a root entry point of the syntax which is labeled as *QUERY*. The *SELECT* and the *FROM* clauses are the only mandatory constructs in

SARI-SQL and therefore they must appear in the abstract syntax tree. The *WHERE*, *OVERCORR*, *ORDER BY*, *GROUP BY* and *HAVING* clauses are optional but they must appear in the above stated order of the rule *sariSQL*.

The following example pretty much covers the most important constructs and is for the purpose of illustrating the top level AST structure shown in Figure 6.15. For the sake of completeness the clauses *GROUP BY* and *HAVING* are shown as well as dotted boxes as they don't appear in the provided example. In the following section each sub-AST will be introduced, described, brought into context with EAExpressions and is explained with examples.

Example:

a) *SELECT start.TransportID, start.StartLocation, end.EndLocation*
FROM TransportStart start, TransportEnd end
OVERCORR TransportInfo
ORDER BY start.StartLocation ASC

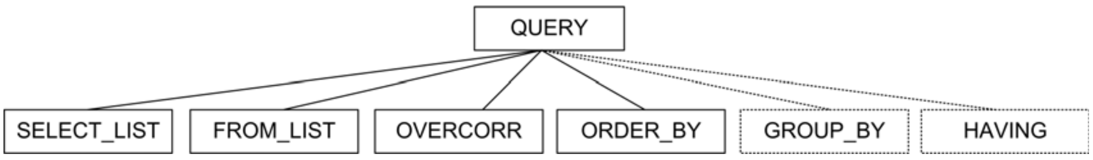


Figure 6.15: Overall SARI-SQL AST representation

6.3.2.2 FROM Clause AST Representation

6.3.2.2.1 FROM Construct

fromConstruct ⇒	*fromList* ∨
	joinClause
fromList ⇒	*FROM_LIST (fromItem)**
fromItem ⇒	*FROM_ITEM reducedDotExpr (fromAlias)?* ∨
	SPECIAL_FROM_ITEM ϕ *(fromAlias)?*

The *FROM* AST structure is defined by the *FROM_LIST* that is a top level node. This node contains all comma separated items which define the event object types and the metrics/scores for the query. Each reference to an event object type is encapsulated by a *FROM_ITEM* node, where the left child node is the type identifier and the right node is the corresponding alias. The alternative rule, of the *fromItem*, defines the tree structure for metrics and scores hold by the node *SPECIAL_FROM_ITEM*. The ϕ element contains the structure for a function whose name-identifier is either *Metric* or *Score*. The parameter list is restricted to one parameter that defines the name of the corresponding type. Mentioned in the previous section, it is always necessary to alias the types if there is more than one item in the *FROM* clause.

In the following SARI-SQL example all attributes of the event object types *Transport-Start* and *TransportEnd* are selected. Furthermore, the metric *AvgTransportDuration* is included. The corresponding AST representation for this selection is illustrated in Figure 6.16

Example:
a) *SELECT * FROM TransportStart start, TransportEnd end, Metric('AvgTransportDuration') mAvg*

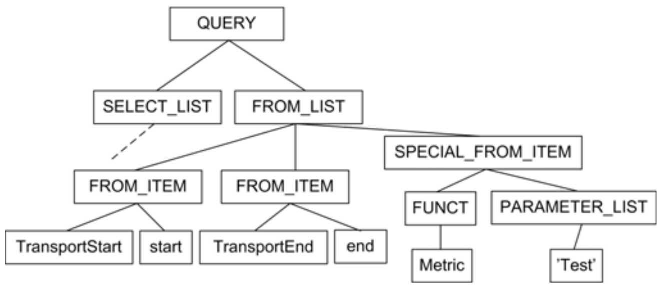


Figure 6.16: Simple FROM clause AST representation

6.3.2.2.2 Join Constructs

The advanced join constructs *INNER JOIN*, *LEFT/RIGHT/FULL OUTER JOIN* in the *FROM* clause are represented by the *joinClause* rule which is nested as an alternative in the *fromConstruct*. In contrast to normal *FROM* expressions the top level node *FROM_ITEM* is replaced by the corresponding join type (e.g. annotation for outer join, inner join, ...). So for instance a full outer join is represented by the node *FULL_OUTER_JOIN*. The first two left child nodes represent the two event object types that are joined including also their alias. The third child element holds the comparison expression that determines on which attribute the join is performed independent from the type of the join.

joinClause ⇒	*INNER_ JOIN FROM_ ITEM FROM_ ITEM onJoinClause* ∨ *FULL_ OUTER_ JOIN FROM_ ITEM FROM_ ITEM fullOuterJoinClause* ∨ *LEFT_ OUTER_ JOIN FROM_ ITEM FROM_ ITEM leftOuterJoinClause* ∨ *RIGHT_ OUTER_ JOIN FROM_ ITEM FROM_ ITEM rightOuterJoinClause* ∨

The following query performs a full outer join over the *TransportID* attributes of the event object types *TransportStart* and *TransportEnd*. The resulting AST is illustrated in Figure 6.17. The *ON_ JOIN_ CLAUSE* represents the *ON* conditional part of the join. This AST tree contains an EAExpression dot notation that can be combined with the operators OR, AND or EQUAL. In the example an equals-join is performed over the *TransportID* attribute. Every other join construct (inner, left/right outer join) in SARI-SQL is represented in the AST the same way. The only difference is always the label of the join node (e.g. instead of *FULL_ OUTER_ JOIN* it contains *LEFT_ OUTER_ JOIN*, *RIGHT_ OUTER_ JOIN* or *INNER_ JOIN*.

Example:

a) *SELECT * FROM TransportStart start FULL OUTER JOIN TransportEnd end ON start.TransportID = end.TransportID*

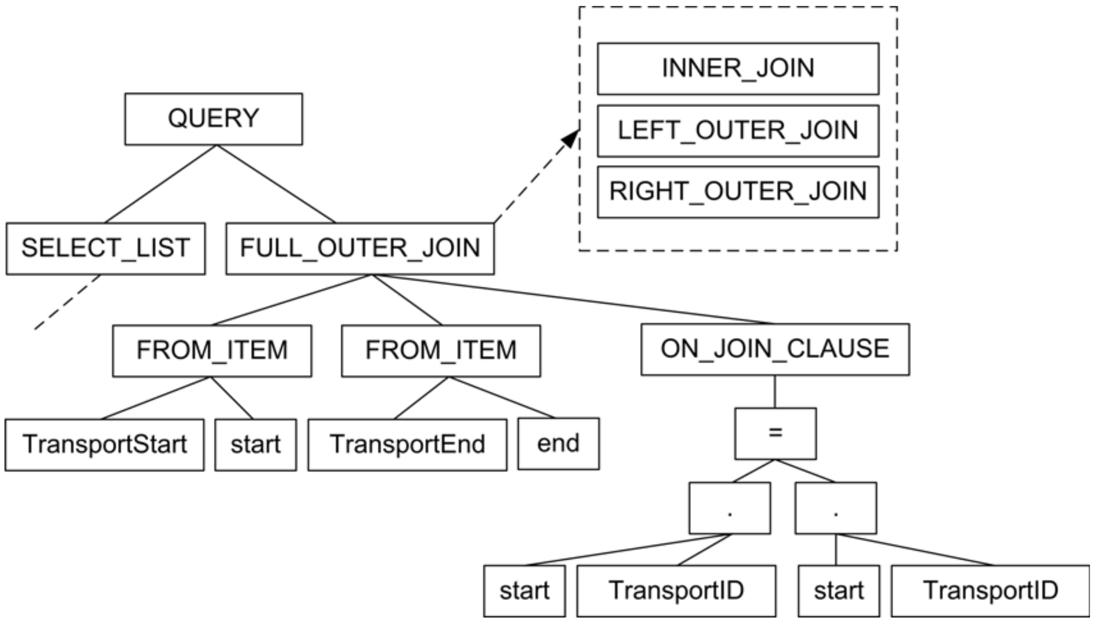


Figure 6.17: Outer Join construct in FROM clause AST representation

6.3.2.3 SELECT Clause AST Representation

The *SELECT* clause consists of a list of one or more items separated by a comma, from a syntactical point of view, where each of the items is a projection that is represented as a column in the result. The AST representation of a *SELECT* clause is a set of *SELECT_ITEM* child nodes, assigned to the top level *SELECT_LIST* node. The items themselves are EAExpression ASTs that are hung to the *SELECT_LIST*. In case that aliasing is used in the *SELECT_LIST* then the outermost left node is always the aliasing identifier in the tree. This special case is addressed during the evaluation of the query and will be explained in Section 6.3.3. Labels that are applied to an item are represented as the right child node of a *SELECT_ITEM*. An important challenge here is the EAExpression validation and evaluation. In EAExpressions the outermost left identifier in dot notations is referring to the event object type identifier.

selectClause $\Rightarrow$	*SELECT_LIST (selectItem)**
selectItem $\Rightarrow$	*SELECT_ITEM* ϕ *(displayName)?*
displayName $\Rightarrow$	*STRING_LITERAL* $\vee$ *IDENTIFIER*

The Figure 6.18 shows the AST representation of the query example below. This example covers the elements of aliasing, shows how EAExpressions are nested into the SARI-SQL AST and the position of labels.

Example:

a) *SELECT created.ShipmentID as 'ID', created.FreightValue - created.Costs as 'Diff' FROM ShipmentCreated created*

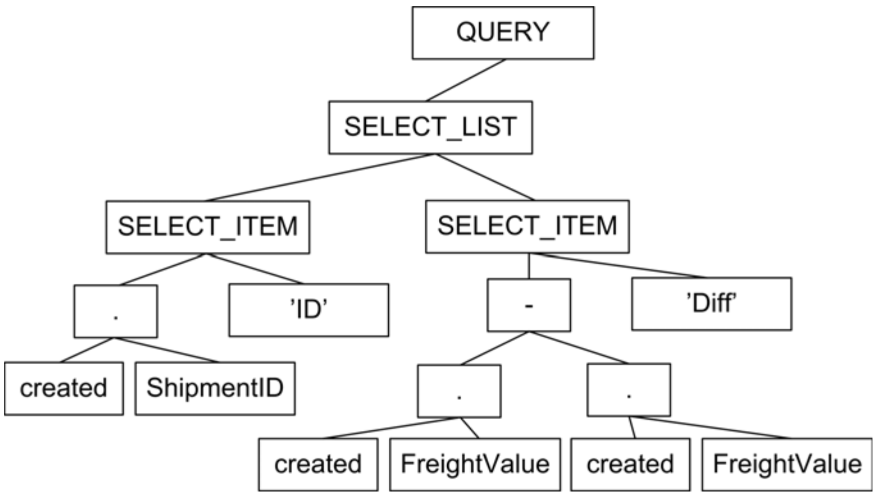


Figure 6.18: SELECT clause AST representation

6.3.2.4 WHERE Clause AST Representation

whereCondition	$\Rightarrow$	*WHERE_CLAUSE boolExpr*
boolExpr	$\Rightarrow$	*LPAREN selectExpression RPAREN* $\vee$
		conditionExpr ((AND $\vee$ *OR* $\vee$ *XOR) boolExpr)?* $\vee$
		NOT boolExpr
boolExpr	$\Rightarrow$	*AND boolExpr boolExpr* $\vee$
		OR boolExpr boolExpr $\vee$
		EQUAL boolExpr boolExpr $\vee$
		ϕ

The *WHERE* clause is the conditional part of an SARI-SQL query and must result in a boolean return value. The sub nodes of the top level *WHERE_CLAUSE* are EAExpression ASTs whereas the syntax rules are overridden and have been slightly modified to enforce boolean return types. Therefore, only arithmetic operations are suppressed and result in an error. The following example filters out the events that don't have *Vienna* and *Madrid* as start and end locations of transports and the corresponding AST representation is shown in Figure 6.19. As with the *SELECT* clause, if aliasing is used then the outermost left node represents the alias, which results in a complication with the originial dot notation in EAExpressions. The solution to this issue is discussed in Section 6.3.3.

Example:
a) *SELECT * FROM TransportStart start, TransportEnd end*
WHERE start.StartLocation = 'Vienna' AND end.EndLocation = 'Madrid'

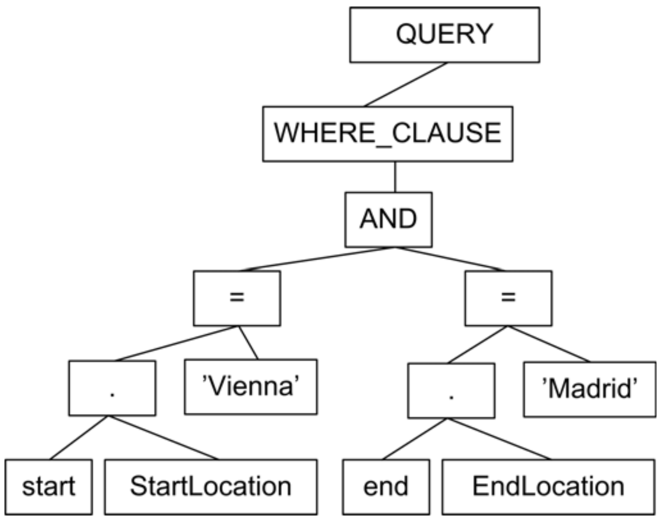


Figure 6.19: WHERE clause AST representation

6.3.2.5 OVERCORR Clause AST Representation

overCorrClause ⇒	*overCorrList*
overCorrList ⇒	*OVERCORR (overCorrItem)**
overCorrItem ⇒	*OVERCORR_ITEM corrName (corrAlias)?*
corrName ⇒	*IDENTIFIER*
corrAlias ⇒	*IDENTIFIER*

The access of correlations, which is a key characteristic of event processing solutions, is in terms of its syntax straight and simple. The keyword *OVERCORR* can be applied after the *FROM* clause and defines the correlation identifer. If more than one item is given in the *OVERCORR* clause list, aliasing must be applied like in the *FROM* clause. Otherwise it is optional. From the AST point of view the representation is similar to *FROM*-items. However, if aliasing is used in correlations, the items (e.g. event object types) in the *FROM* clause can have correlational aliases which mean that the event object type is then selected from the correlation sets.

The following example accesses the *TransportInfo* correlation that consists of the two event object types *TransportStart* and *TransportEnd*. In the *FROM* clause the *TransportStart* event object type is part of the *TransportInfo* correlation (defined through the alias), while every *ShipmentCreated* event is taken into account regardless of correlations. An illustration of the AST representation is shown in Figure 6.20. In reality the query would have to be adapted with conditional expressions to reduce the resultset to a reasonable size in order to create a meaningful result. However a trade off has been made for simplicity reasons and therefore the *SELECT* and *WHERE* clause has been neglected in this example.

Example:

a) *SELECT * FROM infoCorr.TransportStart start, ShipmentCreated created OVERCORR TransportInfo infoCorr*

6.3.2.6 Additional Constructs AST Representation

SARI-SQL supports common SQL constructs such as *ORDER BY, GROUP BY* or *HAVING*. All of these three clauses are represented by top-level nodes that are hung directly as children to the *QUERY* root node. The *HAVING* clause AST construct is represented by the EAExpression *boolExpr* AST, while the ORDER BY and GROUP BY have almost similar representations.

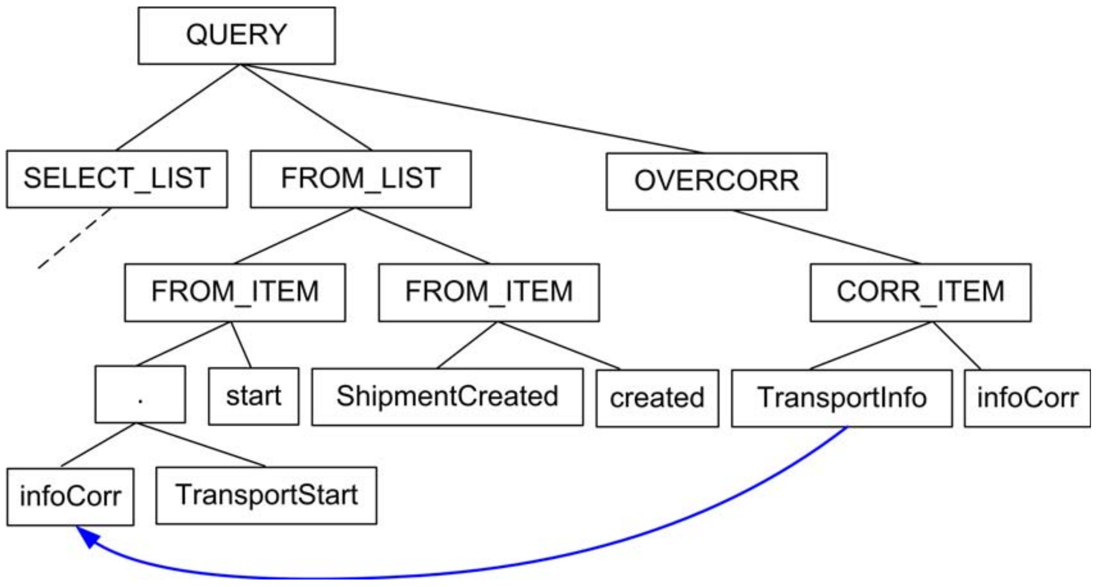


Figure 6.20: OVERCORR clause AST representation

orderByCondition	$\Rightarrow$	*orderByList*
orderByList	$\Rightarrow$	*ORDERBY (orderByItem)**
orderByItem	$\Rightarrow$	*ORDERBY_ITEM reducedDotExpr (ASC* $\vee_C$ *DESC)?*
groupByCondition	$\Rightarrow$	*groupByList*
groupByList	$\Rightarrow$	*GROUPBY (groupByItem)**
groupByItem	$\Rightarrow$	*GROUPBY_ITEM reducedDotExpr*
havingCondition	$\Rightarrow$	*HAVING boolExpr*

The AST of the following examples are represented in Figure 6.21.

Example:

a) *SELECT TransportID, StartLocation, Type*
FROM TransportStart
ORDER BY StartLocation ASC

b) *SELECT TransportID, StartLocation*
FROM TransportStart
GROUP BY StartLocation

c) *SELECT ShipmentID, TransportInfo.Destination, Sum(Costs)*
FROM ShipmentCreated
GROUP BY TransportInfo.Destination HAVING Sum(Costs) > 100

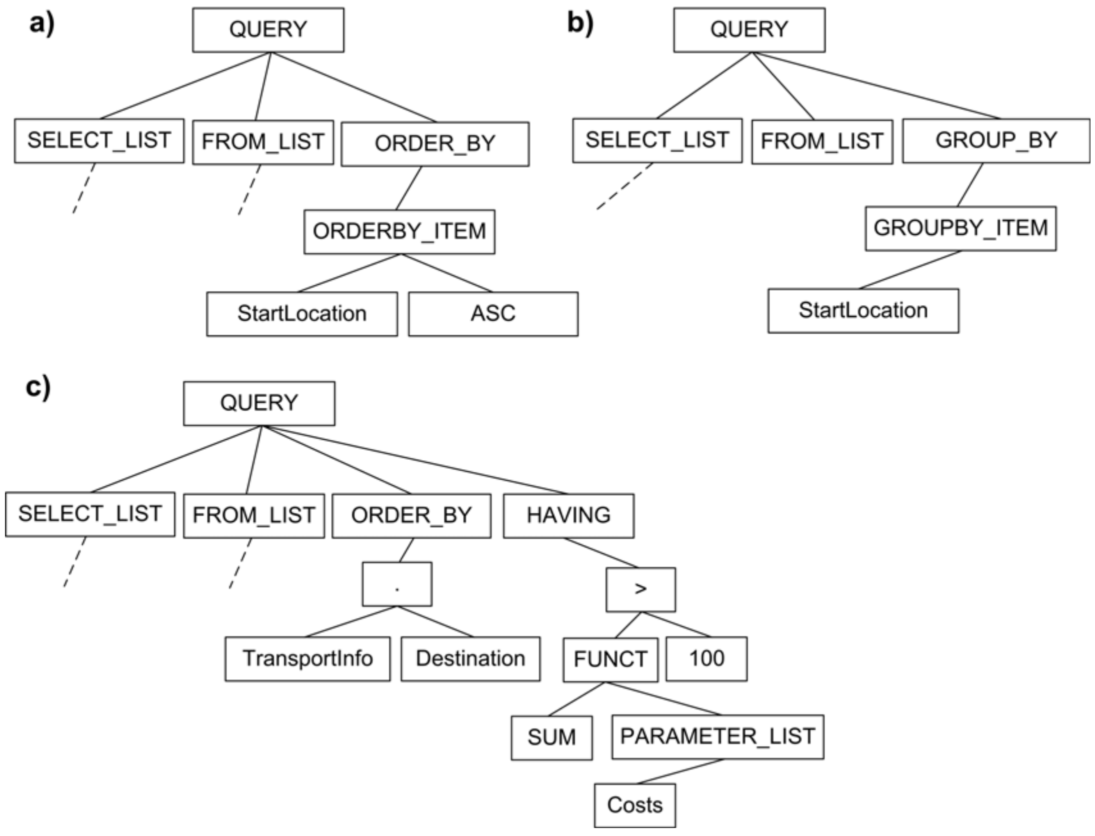


Figure 6.21: Additional Constructs AST representation

6.3.3 SARI-SQL Evaluation Model

The evaluation process of SARI-SQL is split up into four stages where, in contrast to EAExpressions, the validation and evaluation process is not explicitly divided from eachother. The four stages of the SARI-SQL query engine processes are introduced subsequently in this chapter. The concrete implementation is not in scope of this thesis, therefore the query engine is described from a conceptual point of view to create a separation from implementation details. However some challenging issues in the implementation and their solutions will be discussed at the end of this section. A short overview is provided hereafter:

- **Query Input:** The query input consists of an API to access and propagate the query statement to the underlying SARI-SQL query engine. Further it manages and returns syntax, semantic and evaluation (e.g. query execution) error states, query engine messages (e.g. "explain") for tracking the internal query processing

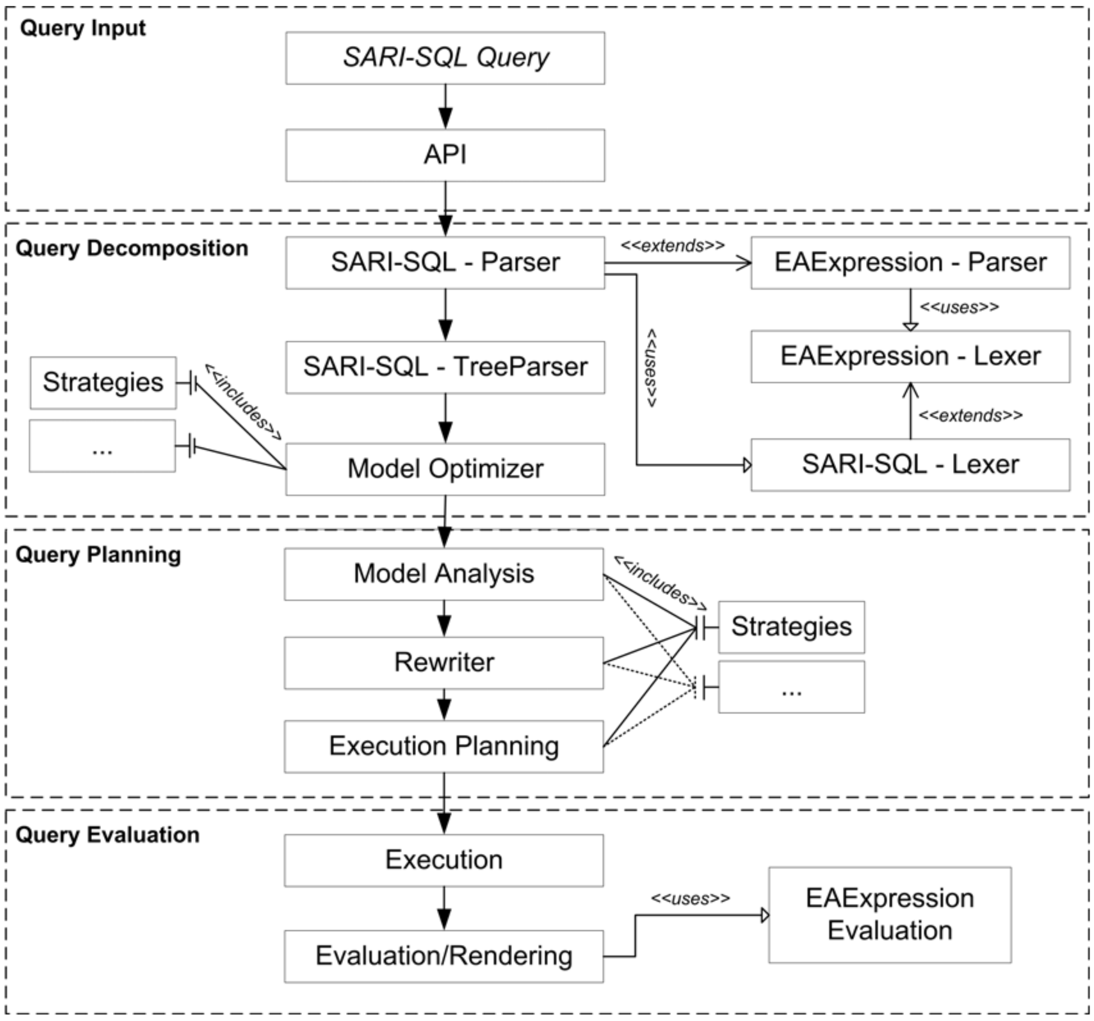


Figure 6.22: SARI-SQL Query Processing Overview

steps and the query result retrieval.

- **Query Decomposition:** The query decomposition covers the SARI-SQL parser, which partly uses the EAExpression Parser rules, that builds an abstract syntax tree out of a given query. The tree parser walks through the syntax tree, checks the correctness of event object types, their attributes, typing correctness and transforms the abstract syntax tree into an internal model representation. The transformed model is then used for query optimization whereas various strategies can be implemented and attached to the query optimizer.

- **Query Planning:** The query planner components analyzes the query model, which was previously assembled and optimized in the query decomposition stage, and performs a rewriting of the query into the target Event-Base repository schema. The rewriting itself adapts the query to the target RDBMS system of the Event-Base and is therefore database vendor independent. The analyzer components further determine the execution strategy whose information is required to post processing the result set.
- **Query Evaluation:** The query evaluation block is in charge of executing the prepared and pre-processed SARI-SQL query. Due to the analysis and rewriting strategies in the previous Query Planning processing steps, it is necessary to reevaluate the returned results from the Event-Base in order to deliver the required results. The EAExpressions evaluation processes are partly used in this stage. Therefore this SARI-SQL processing block can also be seen as a post-processing step before the result is turned over to the API.

The SARI-SQL validation processes consists a syntax parsers, that checks the syntactical correctness of a given query, and the abstract syntax tree parser, that handles the semantic correctness of the query. The semantically correctness of a query checks for instance if the accessed event object types and their attributes exist in the Event-Base. The query validation processes is more comparable to a prepared statement from the Java-World. The SARI-SQL query engine prepares the query for execution by transforming the given query from an abstract syntax model into a more efficient internal data structure that is then used to perform optimization strategies and apply more optimal query rewriting methods. This prepared query model is then available and can be reused during runtime for several executions. If a specific query has to be reused more than once it can save significant time during execution.

The preceding sections will introduce the four SARI-SQL evaluation blocks and the query processing from a conceptual point of view.

6.3.3.1 Query Input

The query input block of the SARI-SQL evaluations processes is schematically illustrated in Figure 6.23 in order to highlight the main access components. Further the flow and usage of the query propagation and the result retrieval is illustrated in the same figure and explained in this section. The SARI-SQL query input consists of an API interface *EB Query Executor API*, which must be parameterized in order to point to the desired Event-Base (compare with Section 5). The *EB Query Executor API* provides the two methods *PrepareQuery* and *ExecuteQuery*.

- **PrepareQuery:** The *PrepareQuery* interface takes a SARI-SQL string and triggers the validation procedures which parses the query, builds up an abstract

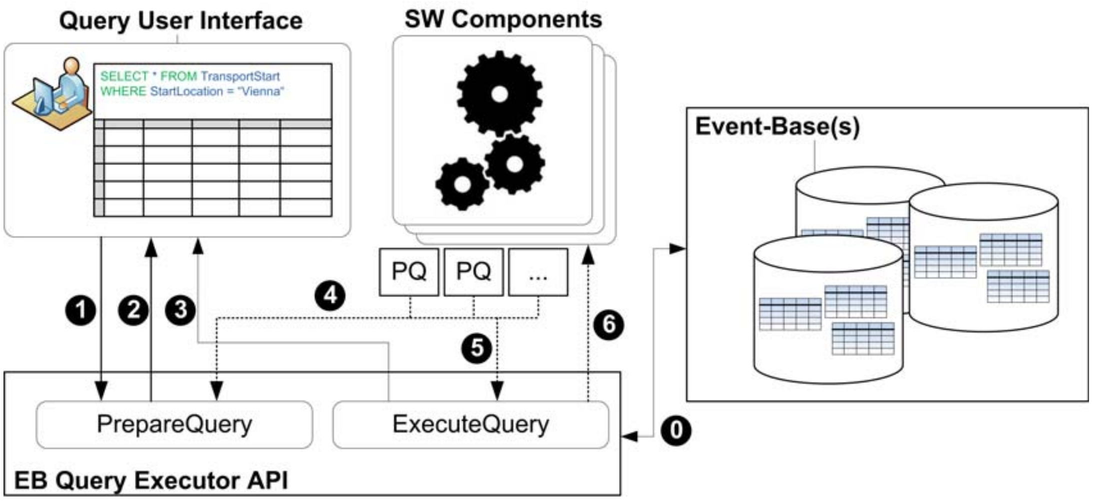


Figure 6.23: SARI-SQL Query Input

syntax tree, reworks that tree into a more efficient internal data representation and then performs optimizations techniques. These are the underlying SARI-SQL steps that are covered by the SARI-SQL query processing blocks Query Decomposition. From the API point of view this interface takes a string and returns an object of the class that contains a detailed listing of errors including the line and character position in case a syntactical error happened or if an event object type was misspelled. After successfully (e.g. without errors) completing this procedure the SARI-SQL query statement is pre-processed, optimized and ready for execution.

- **ExecuteQuery:** The *ExecuteQuery* interface executes the evaluation of the previously prepared and validated query against the selected Event-Base. The underlying SARI-SQL query processing blocks that perform these tasks are the Query Planning and the Query Evaluation. These blocks apply basically analysis and rewriting strategies and then reevaluate the returned results from the Event-Base. From the API point of view this interface requires a validated SARI-SQL query and returns a DataTable containing the results of the query in a column/row manner where the columns represent the items defined in the *SELECT* clause of a query. Further the API allows to retrieve query engine messages for explaining and tracking the internal query processings steps such as the rewriting of the query that is propagated to the underlying RDBMS system and error messages.

Basically the main accessors of the API can be split into two groups. The first one is a human user that is placing queries through a graphical user interface, that supports the query creation with syntax auto-completition and highlighting, and retrieve a graphical representation of the results. The second group are software components that place queries through the *EB Query Executor API* to the Event-Base and process the results further on. Those two groups are represented at the top of the Figure 6.23.

In the first case, the query user interface, the user places a query and triggers the *PrepareQuery* to retrieve potential errors (1). Usually there are always roundtrips, caused by errors, until a query is in a valid state. The status of the query (e.g. an error occurred or the query is valid) is returned to the user interface and prepared for a visual representation (2). If the query is in a valid state the user can execute the query against the Event-Base that returns the query results to the user interface which sets up tabular representation (3).

In the second case, the software components, the query execution steps are the same as with the user interface, with the difference that such software components or modules might often place the same queries several times during runtime of an application. Therefore it is possible to prepare a collection of queries for execution by using the *PrepareQuery* and hold the validated and pre-processed state (4). These Prepared Queries (PQ) are represented by rectangles below the SW Components block in Figure 6.23. During execution runtime these prepared queries can be executed (5) much faster as several preparation steps are dispensed. The results (6) are returned as DataTable objects (6) and can be further processed by the components.

From a certain point of view the user interface is also a software component that is reusing the SARI-SQL results for graphical representation. However the separation in this illustration is done for the reason that user interfaces are for explorative purposes, which usually result in several roundtrips due to errors. SW components however must already use a valid query to avoid errors which is achieved through pre-validating them through a GUI by a developer. In either case the *EB Query Executor API* must be initialized by providing the parameter that sets the pointer to the desired Event-Base.

6.3.3.2 Query Decomposition

The query decomposition block of the SARI-SQL evaluations processes is schematically illustrated in Figure 6.24. The main job of the decomposition block is to prepare a given SARI-SQL query statement for execution (e.g. the query is pre-processed, optimized and made ready for execution against an Event-Base) and to catch errors within the query. The decomposition block consists of the SARI-SQL parser, that

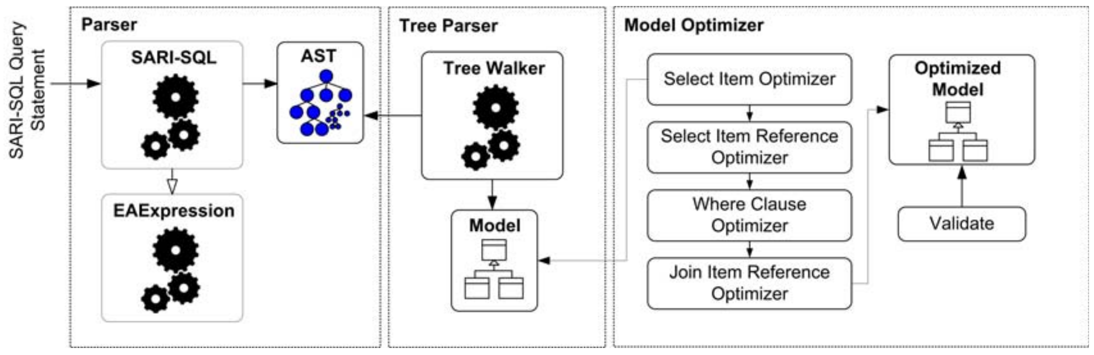


Figure 6.24: SARI-SQL Query Decomposition

creates an abstract syntax tree out of a given query. An abstract syntax tree walker traverses through the previously created AST and validates the correctness of the event object types, their attributes, checks the typing correctness and transforms the abstract syntax tree into an internal model representation. This section describes the query decomposition process step-by-step whereas the internal model creation and the optimization procedures are the most interesting topic for discussion.

6.3.3.2.1 Parser

The query parser, extends the EAExpression syntax rules, introduced in Section 6.2.1, by the SARI-SQL syntax rules, described in the Section 6.3.1, wheras certain rules are overridden to handle specific characteristics that apply to SARI-SQL, but are allowed with EAExpressions. So for instance the *WHERE* clause overrides the rule *boolExpr* to omit the creation of arithmetic resulting operations in that clause while every other EAExpression feature and function is available.

In Figure 6.24 the query input, the SARI-SQL query statement, enters the parser realm from the left side. The input itself is triggered by the *PrepareQuery* interface of the *EB Query Executor API* in the query input block. The parser builds up an abstract syntax tree by making use of the SARI-SQL syntax definition rules. The AST creation processes is completely handled by ANTLR and uses the defined rule declarations of SARI-SQL and the EAExpressions. The basic tree building concepts of ANTLR have been introduced in the Section 6.1. The formal definition of the resulting tree constellations are defined in the Section 6.3.2.

If the parser encounters any syntactical errors it throws an error message with a detailed corresponding description about the encountered problem including the line and character position of the syntactical error. The error message is passed back to the query input as a result for further investigation by the user.

6.3.3.2.2 Tree Parser

The SARI-SQL tree parser is one of the most interesting parts of the query processing. After the parser spanned up an abstract syntax tree, the tree parser or walker traverses through the tree. The way the traversing is done is again defined declaratively by the rules introduced in 6.3.2. ANTLR uses those rules to walk through the tree and if certain rules match specific AST patterns then actions can be executed on those tree elements. The rules that match tree patterns, trigger the creation processes of the internal model representation of the query. Every rule evaluates the patterns and their meta-data, perform checks and set up a model.

The interface definition of the model is shown in Figure 6.25. Every main SARI-SQL clause is represented by a class definition. All four classes *SelectClause*, *FromClause*, *CorrelationClause* and *WhereClause* are attributes of the *SARISQLModel* and implement a BaseCollectionModel abstract class that allows to manage the items of those clauses such as the select or from items that are comma separated. Every class contains the whole corresponding original AST node and service methods such as the infix generator in the *WhereClause*. Subsequently the model creation processes of the model will be explained.

Select Clause

The Figure 6.26 shows the simplified processes of how select items for the *SelectClause* model are generated. First, every select item is checked if it contains a star operator. If it is a star operator a *SelectItemStar* is generated. If the select item does not correspond to a star, a normal *ItemAttribute* creation is triggered. Either way both items are added to the *SelectClause* and the aliasing is extracted if there is one applied. The *SelectClause* itself expects as an item in the list an object of the type *ISelectItem*. The structure of the resulting objects are shown in Figure 6.27. The reason why *ItemAttribute* is not derived from *SelectItem* is because the generated *ItemAttribute* implements on a higher abstraction level the *ISelectItem* (compare with Figure 6.29). The select item can be labeled, which is extracted and stored into the model, if there is one available.

Item Attribute Creation

The specialty about the *ItemAttribute* is that it makes a distinction between

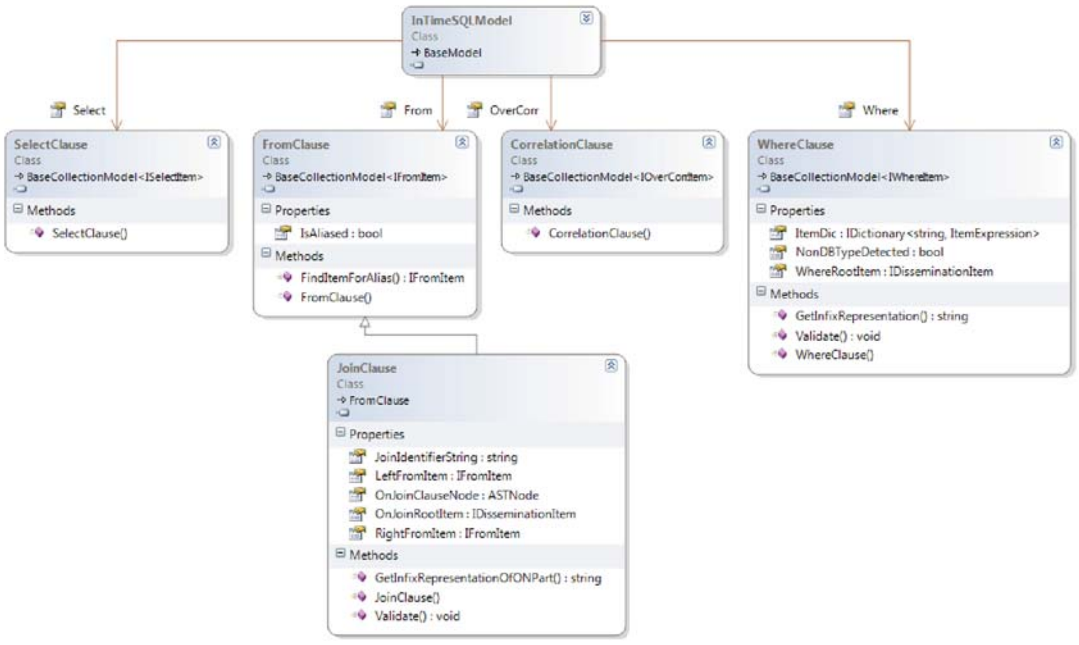

InTimeSQLModel
Class
BaseModel
Select
From
OverCorr
Where
SelectClause
Class
BaseCollectionModel<ISelectItem>
Methods
SelectClause()
FromClause
Class
BaseCollectionModel<IFromItem>
Properties
IsAliased : bool
Methods
FindItemForAlias() : IFromItem
FromClause()
CorrelationClause
Class
BaseCollectionModel<IOverCorrItem>
Methods
CorrelationClause()
WhereClause
Class
BaseCollectionModel<IWhereItem>
Properties
ItemDic : IDictionary<string, ItemExpression>
NonDBTypeDetected : bool
WhereRootItem : IDisseminationItem
Methods
GetInfixRepresentation() : string
Validate() : void
WhereClause()
JoinClause
Class
FromClause
Properties
JoinIdentifierString : string
LeftFromItem : IFromItem
OnJoinClauseNode : ASTNode
OnJoinRootItem : IDisseminationItem
RightFromItem : IFromItem
Methods
GetInfixRepresentationOfONPart() : string
JoinClause()
Validate() : void

Figure 6.25: Class Diagram of the SARI-SQL Model

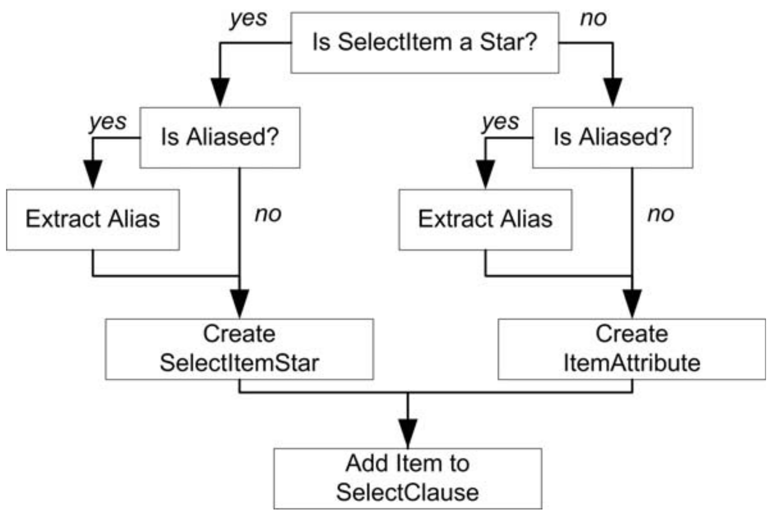


Figure 6.26: Select Item Generation

normal identifiers and EAExpressions. Figure 6.28 shows the item creation processes that is reused also in the *WHERE* and in the *ON* clause. The first step is that it

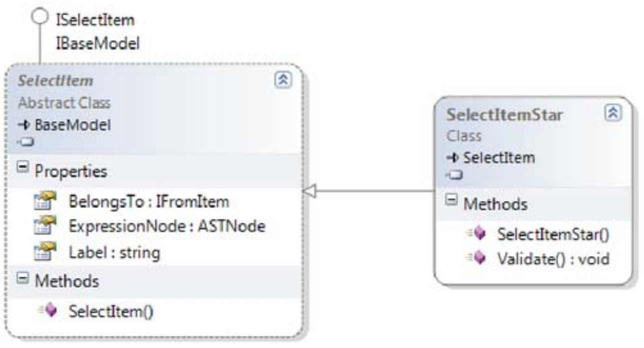


Figure 6.27: Select Item Class Diagram

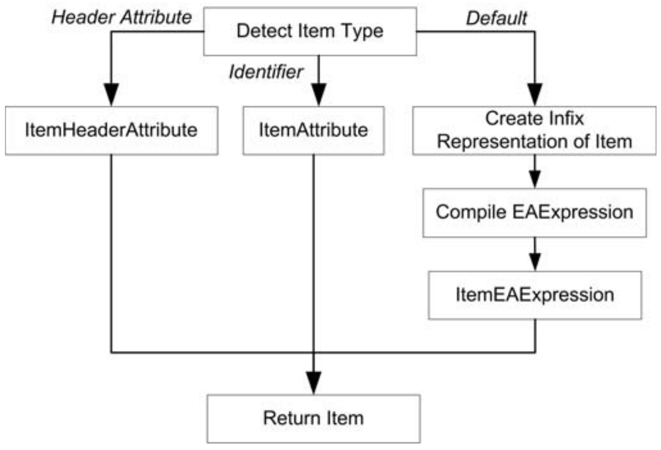


Figure 6.28: Item Attribute Generation

determines the type of the item AST. If it is a header attribute the corresponding *ItemHeaderAttribute* is created and in case of an identifier an *ItemAttribute* is instantiated. However if none of them applied the AST nodes representing that item are transformed into an infix representation and than compiled by the EAExpression parser. The compiled EAExpression statement is then used to create an *ItemEAExpression*. These specialization of classes is alter on used to distinct the item types in the model during query rewriting and the post-evaluation phase.

FROM Clause

The Figure 6.30 shows the processes of how from items are created. The given AST node is checked for whether what type of source it is. The tuple source can be either an event object type, a metric or a score. In case of an event object type, the

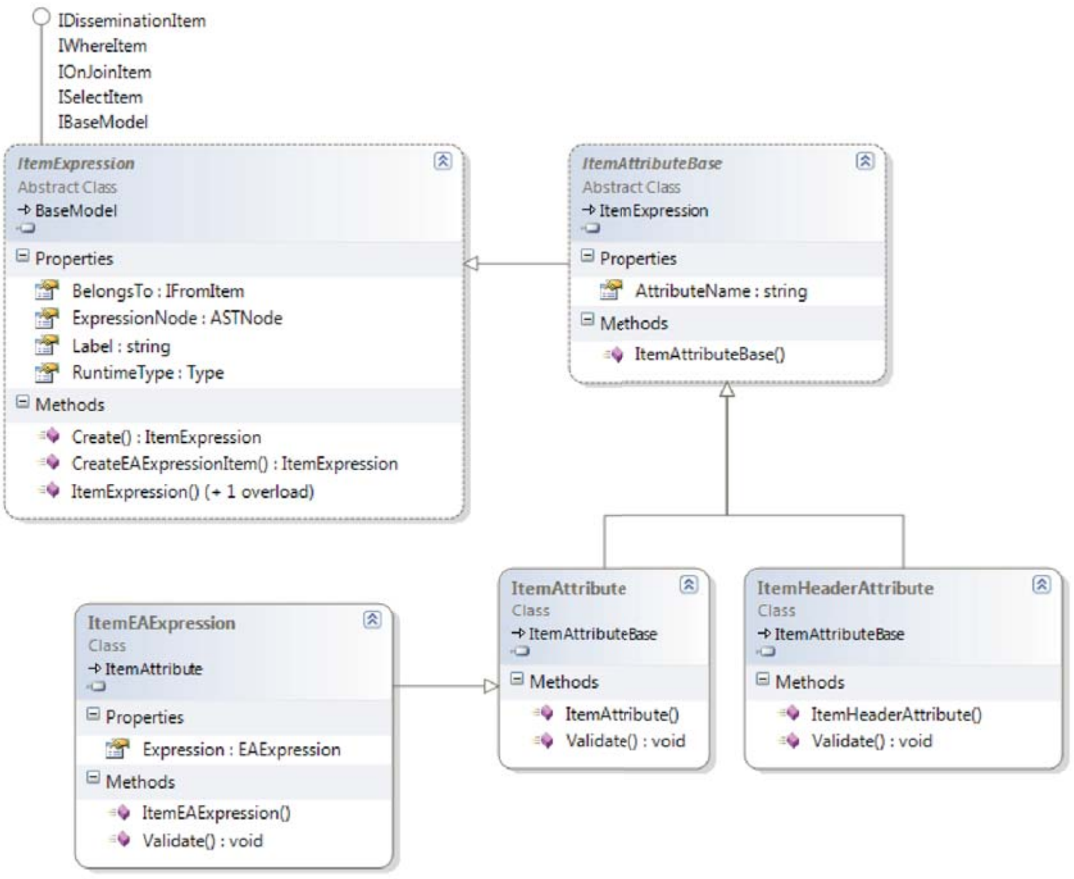


Figure 6.29: Item Expression Class Diagram

corresponding type is resolved from the Event-Base and a *FromItemTable* object is created containing the event object type information from the EOT library. In case of the special constructs of metrics and scores, dummy event object types are generated with the attributes corresponding to the classifier types of the selected scores and metrics. The result of the processes is a *ScoreFromItem* or *MetricFromItem* object. The aliases of the items are extracted if they are available and at the end the created item model objects are added to the *FromClause*. Figure 6.31 shows the class diagram of the *FromItem*. The specializations are used later on during the rewriting phase and post-evaluation to distinct between the different types just like with the item attributes.

JOIN Clauses

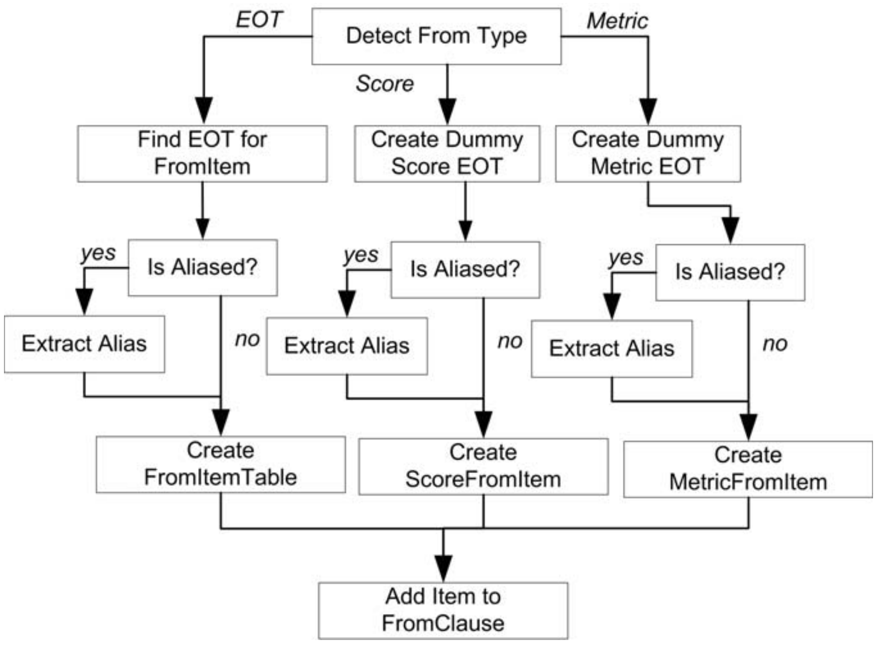


Figure 6.30: From Item Generation

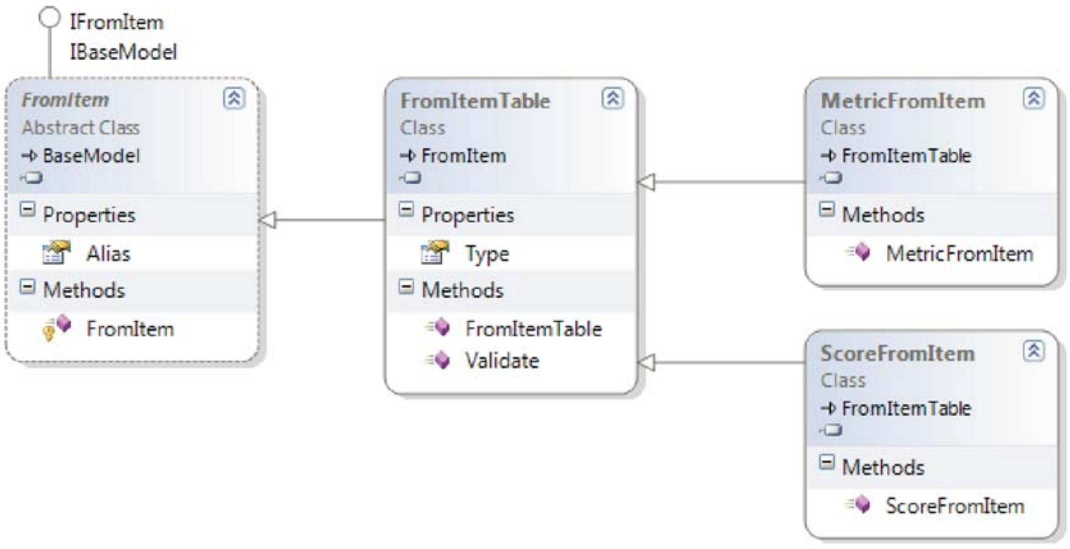


Figure 6.31: FromItem Class Diagram

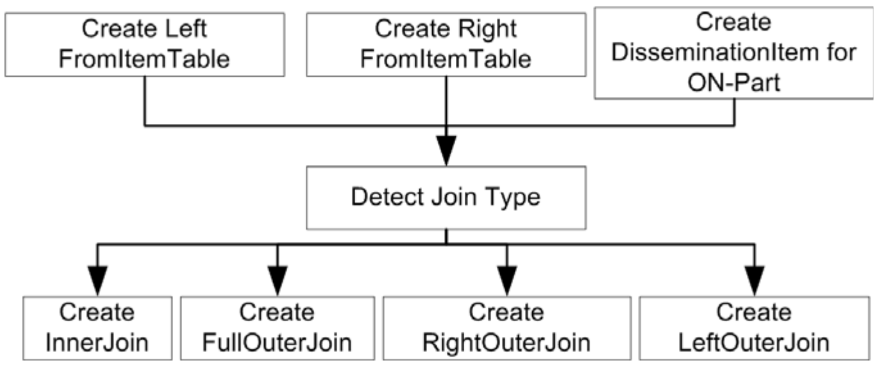


Figure 6.32: Join From Item Generation

The join clause fragments are defined in the *FROM* clause of SARI-SQL and the corresponding AST, with its meta-data, indicates their existence (compare with Section 6.3.2). A join construct in SARI-SQL, independent of its join-type, must contain two data source items (e.g. an event object type for instance) and an *ON* clause that represents the conditional EAExpression part of the join operation. The tree parser extracts the two items that are joined with each other, labeled as *Left/Right FromItemTable* in Figure 6.32. Based on the join-type (e.g. *InnerJoin, FullOuterJoin, RightOuterJoin* or *LeftOuterJoin*) the corresponding instance of a derived *JoinClause* is created. Those instances contain the two event object types that are joined with each other and the *IDisseminationItem* that represents the recursive structure of the conditional clause. The dissemination item process is described in the next paragraph about the *WHERE Clause*. The class diagram representing the basic structure of joins is shown in Figure 6.33.

WHERE Clause

The *WHERE* clause of SARI-SQL contains an EAExpression that must result into a boolean return value and therefore only arithmetic operations are suppressed in that clause. Currently the model creation process strategy performs a recursive decomposition of the *WHERE* clause AST where the breakdown is performed at *OR, AND* and *EQUALS* operator nodes. These decomposition points are important later during the rewriting phase. The rewriter can use those points to optimize the execution strategy. For instance when an equals operator is applied in the *WHERE* clause the left expression of the operator can be fully evaluated by the database system, while the right part must be partly post-evaluated in memory. Especially this decomposition granularity strategy is an important point of future work that can significantly increase the efficiency of the query engine.

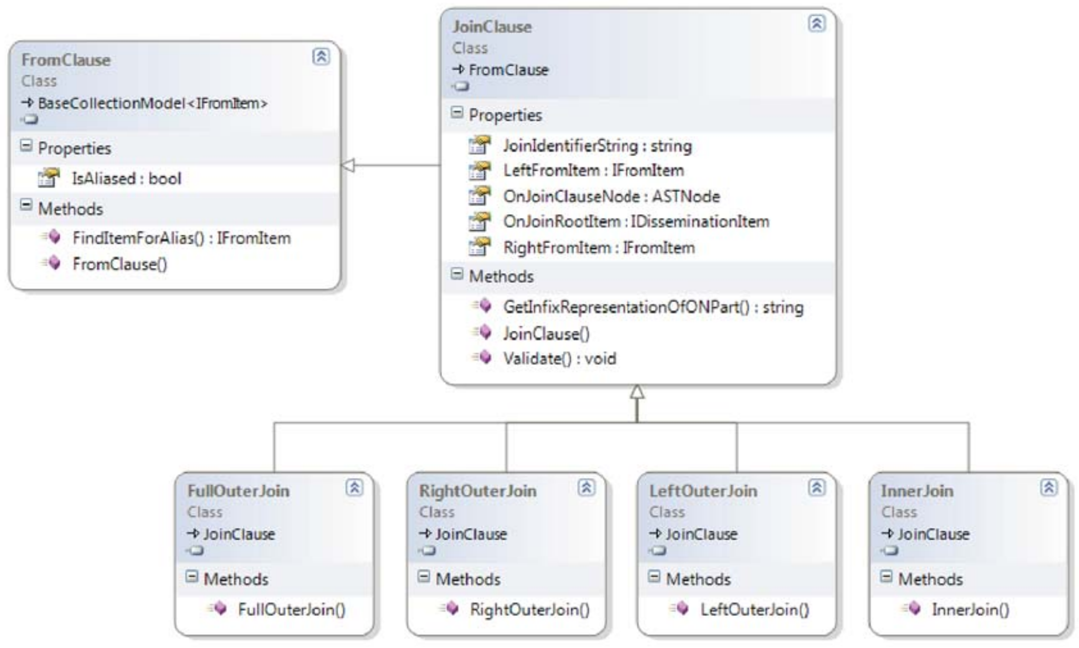


Figure 6.33: FromJoinStructure Class Diagram

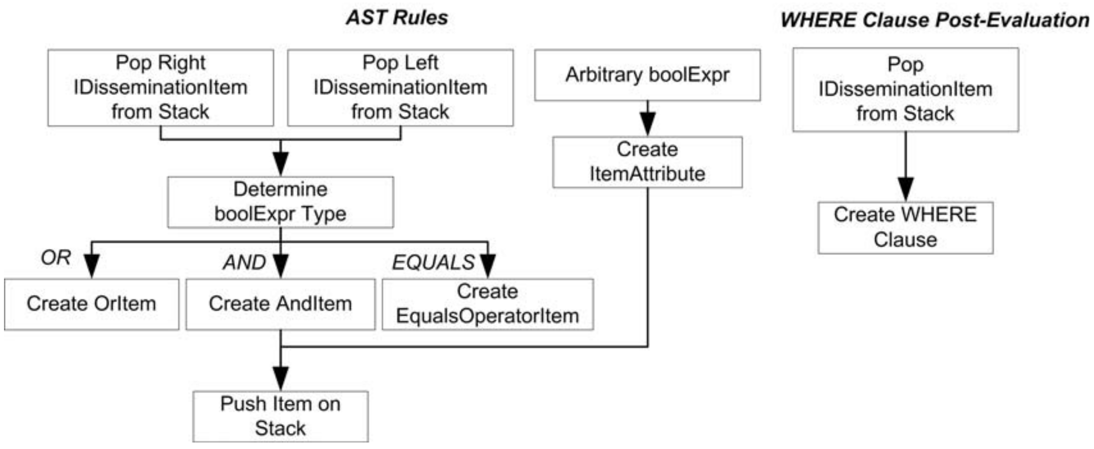

AST Rules
WHERE Clause Post-Evaluation
Pop Right IDisseminationItem from Stack
Pop Left IDisseminationItem from Stack
Arbitrary boolExpr
Pop IDisseminationItem from Stack
Create ItemAttribute
Determine boolExpr Type
Create WHERE Clause
OR
AND
EQUALS
Create OrItem
Create AndItem
Create EqualsOperatorItem
Push Item on Stack

Figure 6.34: Where Clause Dissemination Generation

The dissemination process is illustrated in Figure 6.34. The process is basically based on stack operations where in the processes for arbitrary *boolExpr ItemAttributes* are created and pushed on the stack. As the operators *OR, AND* and *EQUALS* are binary operators the expressions have a left and right part separated by the operator. The corresponding parts are poped from the stack and based on their type the corresponding instances are generated and again pushed on the stack. The class diagram is shown in Figure 6.35. The result of this process is an object compatible to the interface *IDisseminationItem* containing a left and a right *DisseminationItem* part and the concrete implementation determines the operator type.

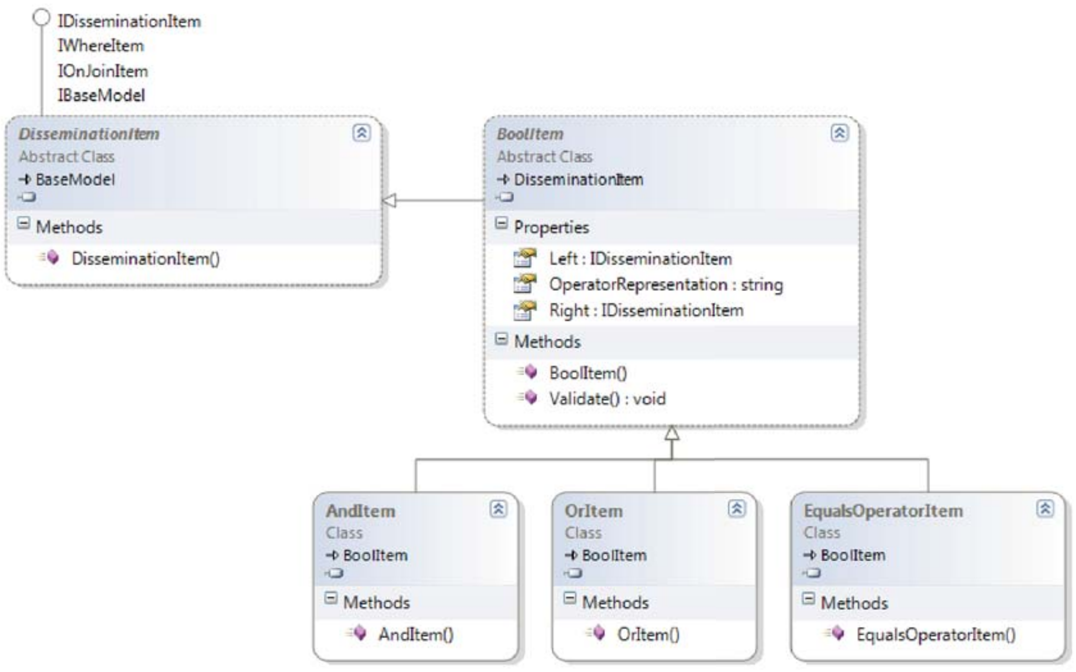


Figure 6.35: DisseminationItem Class Diagram

6.3.3.2.3 Model Optimizer

StarOptimizer

The job of the *StarOptimizer* process (illustrated in Figure 6.36) is to transform the star operator in the *SELECT* clause into concrete *SelectItems* of the corresponding attributes of event object type. The process iterates through the *SelectItems* of the *SELECT* clause. If a star is detected the event object type is tried to resolve and the *ItemAttribute* creation process is performed for every attribute of the event object type. For instance the query *SELECT * FROM TransportStart* is

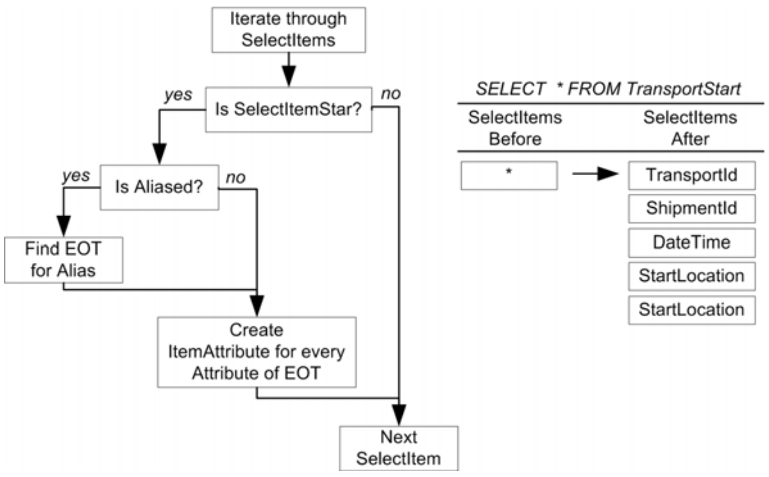


Figure 6.36: StarOptimizer Process

translated internally into all available attributes of *TransportStart*.

ItemAttributeOptimizer

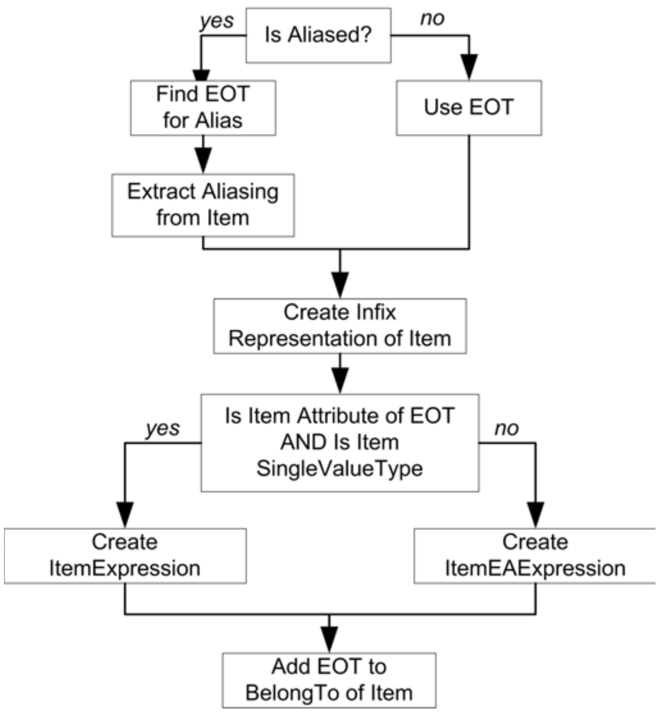

Is Aliased?
yes
no
Find EOT for Alias
Use EOT
Extract Aliasing from Item
Create Infix Representation of Item
Is Item Attribute of EOT AND Is Item SingleValueType
yes
no
Create ItemExpression
Create ItemEAExpression
Add EOT to BelongTo of Item

Figure 6.37: ItemAttributeOptimizer Process

ItemAttributeOptimizer process is reused in several other optimizers as its main job is determine the type of an item. So for instance every item of the *SELECT* clause is checked whether it is a *SingleValueType* attribute of an event object type. This is the first determination if the item can be directly executed against the database. Remember that "flat" structures are directly accessible and more complex attribute types, like nested event object types or dictionaries must be post evaluated (compare with Section 5.5). If the item is not a *SingleValueType* the infix representation is used to create an *ItemEAExpression* by compiling its EAExpression. At last the corresponding event object type of the item is added to the item.

SelectItemBelongToOptimizer

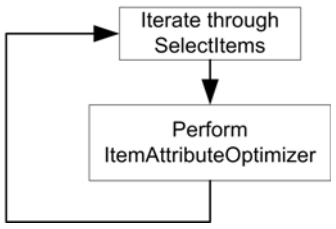


Figure 6.38: SelectItemBelongToOptimizer Process

The *SelectItemBelongToOptimizer* iterates through the *SelectItems* that have been preprocessed by the *StarOptimizer* and applies the *ItemAttributeOptimizer* to all of its items. This process pre-determines if an item is directly accessible on the database or if it has to be post-evaluated. Either way the reference to the corresponding data source (e.g. event object type) is created.

WhereItemBelongToOptimizer and JoinItemBelongToOptimizer

The *WhereItemBelongToOptimizer and JoinItemBelongToOptimizer* apply the *ItemAttributeOptimizer* to the leafs of the *IDisseminationItem*. But the *ON* and the *WHERE* clause contain a root item that has a left and a right item. Those items represent the structure of the conditional expression, possibly containing aliases and expressions that can not be pushed directly to the database. Therefore the optimizer traverses through the model recursevily and applies the *ItemAttributeOptimizer* that creates the link between the corresponding event object type and item and determines if it is an EAExpression that must be post-evaluated.

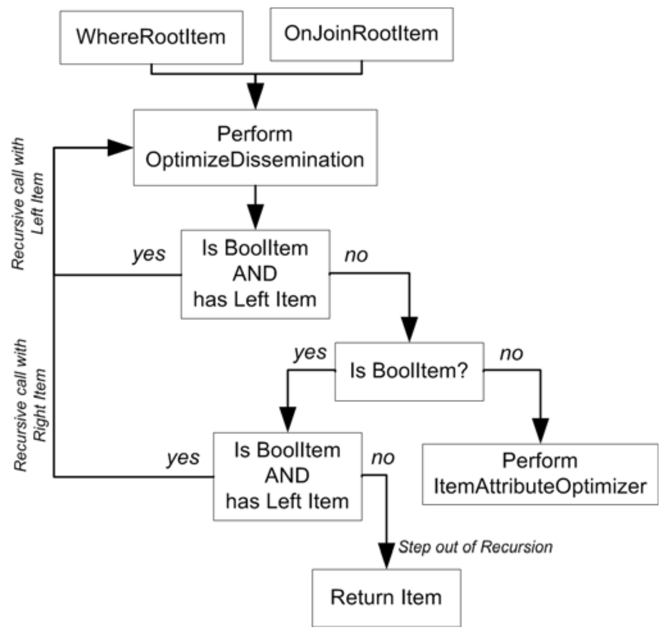

WhereRootItem
OnJoinRootItem
Perform OptimizeDissemination
Recursive call with Left Item
yes
Is BoolItem AND has Left Item
no
yes
Is BoolItem?
no
Recursive call with Right Item
yes
Is BoolItem AND has Left Item
no
Perform ItemAttributeOptimizer
Step out of Recursion
Return Item

Figure 6.39: WhereItemBelongToOptimizer Process

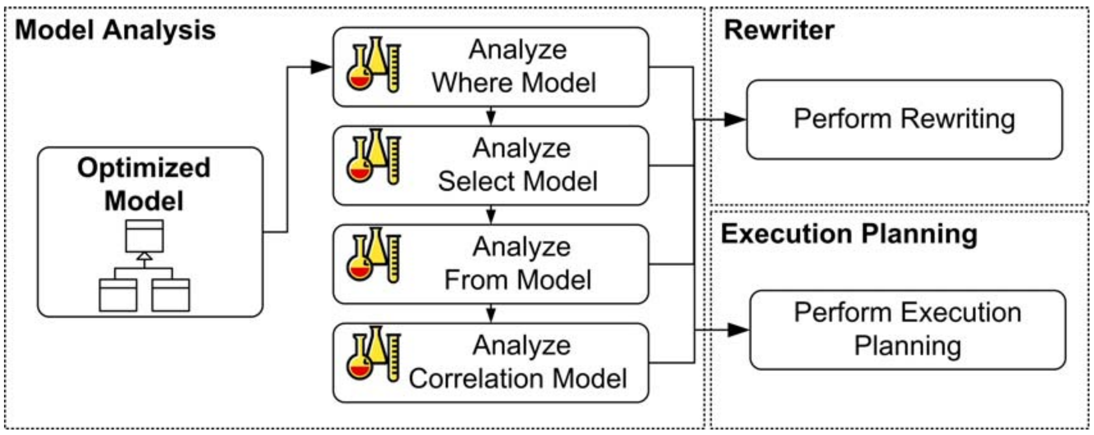


Figure 6.40: SARI-SQL Query Planning

6.3.3.3 Query Planning

In the previous SARI-SQL query decomposition block, the given query statement was pre-processed and optimized by creating an abstract syntax tree, traversing through the AST, validating it, transforming the AST into a more efficient internal data structure and at last applying an optimization strategy to the model. The created and optimized

SARI-SQL query model is now used by the query planning stage for the final processing step before execution. The planning analyses the model, performs rewriting according to the target Event-Base repository schema, taking into account the target RDBMS systems, and setting up an execution plan. The high level process is outlined in Figure 6.40. The process itself takes the SARI-SQL model analyzes the various clauses (Select, From, Where, ...) and performs the rewriting and execution planning based on the results.

6.3.3.3.1 Analyze and Rewrite WHERE Clause

The first SARI-SQL model component that is analyzed is the *WHERE* clause.

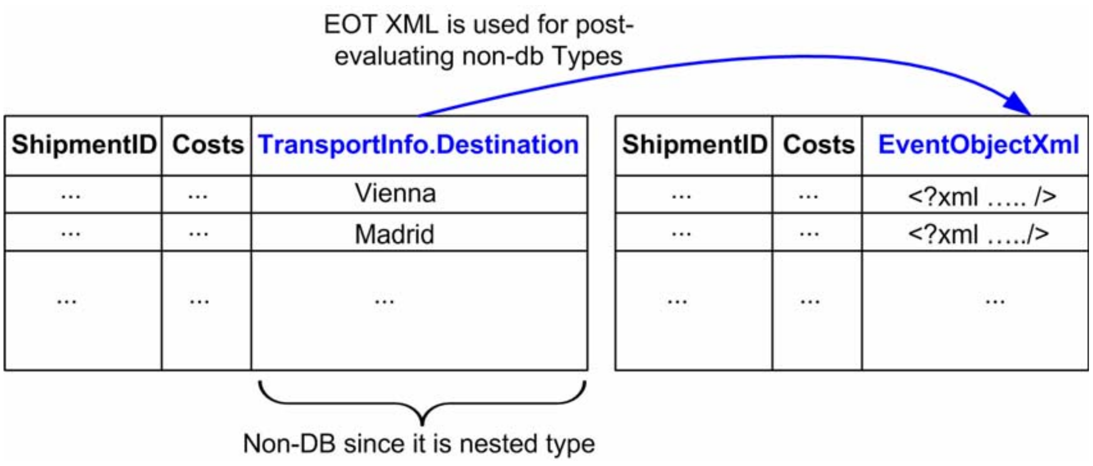

EOT XML is used for post-evaluating non-db Types
ShipmentID
Costs
TransportInfo.Destination
Vienna
Madrid
...
Non-DB since it is nested type
ShipmentID
Costs
EventObjectXml
<?xml ..... />
<?xml ...../>
...

Figure 6.41: Where Clause Analysis and RewritingProcess

As mentioned earlier in the Section 6.3.3.2 the *WHERE* clause AST is broken down at *OR, AND* and *EQUALS* operator nodes, in order to span up at these decomposition points a dissemination of items that now can be used to check if it is possible to push the expressions of the items directly onto the Event-Base. The process is illustrated in Figure 6.44. The *WHERE* clause model has a root item that marks the split point of a *boolItem* (e.g. *OR, AND* and *EQUALS* with the current strategy). Every *boolItem* contains a left and a right *WhereItem* that are iterated through recursevily. If a non *boolItem* is hit, the containing expression of item is checked against the Event-Base if it is representable (e.g. it is a non-DB Type). In case that every item is representable by the Event-Base a rewriting is performed into the target Event-Base repository schema taking the special nature of the applied RDBMS systems into account. However if an item is not representable (such as nested attributes in event object types), a unique label is created for it and stored for further evaluation. After all *WHERE* clause items are checked the list with the non-DB items is taken and they are flagged as non-DB Executables and pushed into the list of *SELECT* clause items. These newly created *SELECT* clause items are used in the post-evaluation process to apply in-memory filtering and joins according to the *WHERE* clause expression part that could not be performed directly by the Event-Base. After the post-evaluation has been successfully finished these type of items are excluded from the projects. items are used in the post-evaluation process

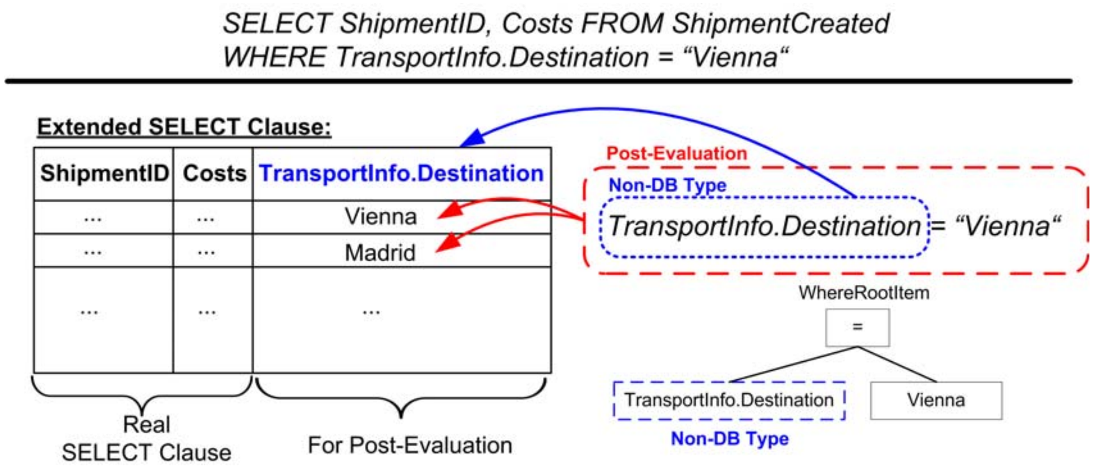

SELECT ShipmentID, Costs FROM ShipmentCreated
WHERE TransportInfo.Destination = "Vienna"
Extended SELECT Clause:
ShipmentID
Costs
TransportInfo.Destination
Vienna
Madrid
Real SELECT Clause
For Post-Evaluation
Post-Evaluation
Non-DB Type
TransportInfo.Destination = "Vienna"
WhereRootItem
=
TransportInfo.Destination
Vienna
Non-DB Type

Figure 6.42: Where Clause Planning Example

Figure 6.44 provides an example to explain the concept of non-DB item handling. In the example a query is executed that selects the *ShipmentID* and *Costs* attributes out of the *ShipmentCreated* event object type. Further the *WHERE* clause restricts the space of tuples by the constraint *TransportInfo.Destination* = *'Vienna'*. The left part of this *WHERE* clause constraint is detected as a non-DB type as it is a nested type. Therefore an additional *SELECT* clause item is added that represents the item expressions that can't be pushed directly to the Event-Base for evaluation. Later on, in the post-evaluation the the newly created column is filled up with the data taken from the corresponding event object type XML (i.e. *ShipmentCreated.Destination*). These tuples are evaluated against the previously non-DB item expressions. The evaluation itself is an EAExpression procedure.

6.3.3.3.2 Analyze and Rewrite SELECT Clause

The *SELECT* clause model consists of several items that were built out of the abstract syntax tree and later on optimized in the query decomposition block. Each of those items represent a projection expression. A projection expression can be in the simplest case just an attribute of an event object type or in a more complex case a calculation that can be defined by EAExpressions. The type of each item (e.g. *ItemHeaderAttribute, ItemAttribute* or *ItemEAExpression* - compare with Figure 6.29) was determined during the transformation processes of the AST to the internal model and is represented through an inheritance hierarchy described in Section 6.3.3.2.

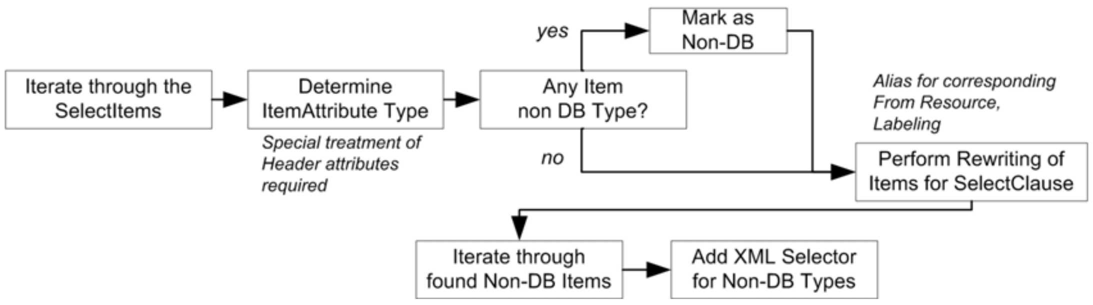
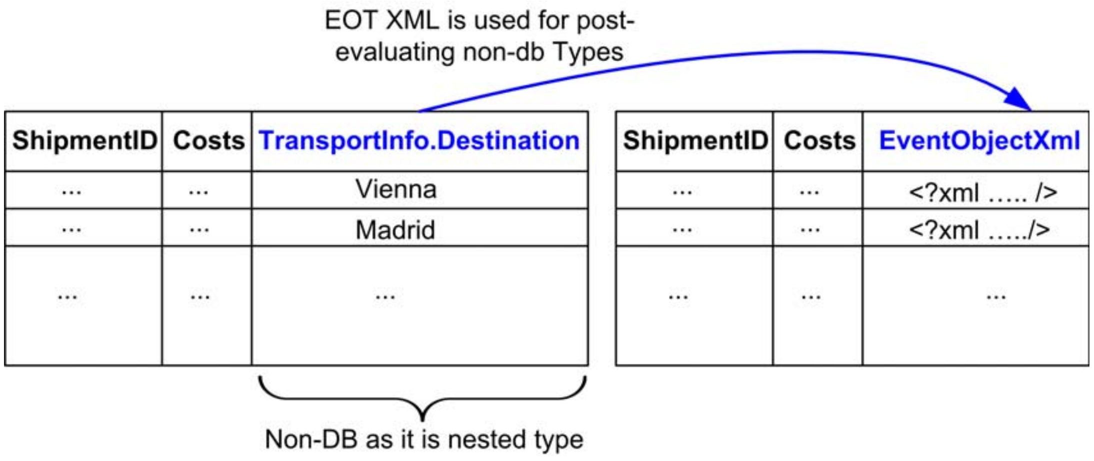

Iterate through the SelectItems
Determine ItemAttribute Type
Special treatment of Header attributes required
Any Item non DB Type?
yes
Mark as Non-DB
no
Alias for corresponding From Resource, Labeling
Perform Rewriting of Items for SelectClause
Iterate through found Non-DB Items
Add XML Selector for Non-DB Types

Figure 6.43: Select Clause Analysis and Rewriting Process

Now these types are required during the the *SELECT* clause planning process as the first step is to iterate over the *SELECT* clause model items. A simplified illustration of the process itself is shown in Figure 6.43. Every select item is checked first, which item type it is - this is required for rewriting purposes to determine the specific characteristics of the item. The next step is to check whether the expression of the particular item is not representable directly by the Event-Base (i.e. it is a non-DB Type). If the item is not representable, the corresponding *FROM* source, referenced in the internal model, is stored to a list for later processing (i.e. BlongsTo attribute of the item). The items that are representable by the database receive in the rewriting process the right alias of their corresponding *FROM* source and transformed into the target Event-Base repository schema taking the special nature of the applied RDBMS system into account. Basically the rewriting process just translates the items with their expressions into the target RDBMS SQL dialect and takes care about the right usage of the aliases.

At last the previously created list of *FROM* items is worked through. The rewriter creates for every source item of that list an additional projection in the target RDBMS SQL dialect. That projection item accesses the *EventObjectXml* attribute of the corresponding type (compare with Section 5.5).

Figure 6.44 shows an example of the processing of non-DB types for better understanding. This example accesses following three items of the event object type *ShipmentCreated* in the *SELECT* clause: *ShipmentID, Costs* and *TransportInfo.Destination.* As the *TransportInfo.Destination* is a nested type it is not possible to push it directly to the database. Therefore the rewriter creates an access to *EventObjectXml* of the *ShipmentCreated* event.


EOT XML is used for post-evaluating non-db Types
Non-DB as it is nested type

ShipmentID	Costs	TransportInfo.Destination
...	...	Vienna
...	...	Madrid
...	...	...

ShipmentID	Costs	EventObjectXml
...	...	<?xml ..... />
...	...	<?xml ...../>
...	...	...

Figure 6.44: Where Clause Analysis and Rewriting Process

To provide a better idea of what the query rewriter creates when dealing with nested types or in general non-DB types, the subsequent SARI-SQL example (*a*) shows the translation (*b*) performed by the query rewriter. In this example the target RDBMS is MS SQL Server 2005.

Example:
a) *SELECT ShipmentID, Costs, TransportInfo.Destination FROM ShipmentCreated*
b) *SELECT dbo.[ShipmentCreated_X].EventObjectXml AS ShipmentCreated_EventObjectXml FROM dbo.[ShipmentCreated_X]*

6.3.3.3.3 Analyze and Rewrite FROM Clause

The *FROM* clause is the construct that defines the data sources that are under examination in the projection (*SELECT*) clause or the condition (*WHERE*) clause. In particular the *FROM* clause consists of several items (compare with Figure 6.31) which defines event object types and the metrics or scores for the query. The internal model of a normal *FROM* clause representation (e.g. without containing join constructs) holds basically the corresponding event object type and an alias that is used through out the rest of the SARI-SQL query. In case of metrics and scores an internal dummy event object type is used to represent their structures. SARI-SQL model items hold references to their corresponding *FROM* items.

In case of advanced join constructs such as *INNER JOIN or LEFT/RIGHT/FULL OUTER JOIN* the model contains (compare with Figure 6.33) two data source items (e.g. an event object type for instance) and an *ON* clause that represents the conditional EAExpression part of the join operation. Just like the *WHERE* clause the *ON* clause is presented through a root item that marks the split point of a *boolItem* (e.g. *OR, AND* and *EQUALS* with the current strategy.

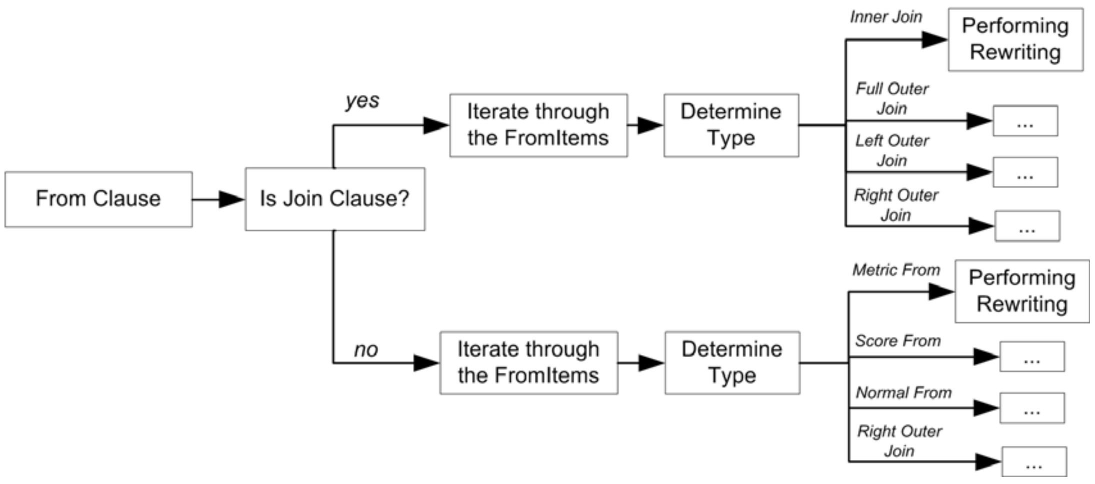


Figure 6.45: From Clause Analysis and Rewriting Process

The simplified process of analyzing and rewriting the *FROM* clause is illustrated in Figure 6.45. This process is pretty simple, in terms of complexity, as it makes a distinction between normal join constructs and the constructs that contain join. Then it iterates over all *FROM* items and, determines the type of the item and applies the corresponding rewriting procedures. The rewriting translates the SARI-SQL *FROM* items into the corresponding RDBMS dialect of the Event-Base and applies the correct aliasing. For the *ON* clause of join constructs currently the same strategies are applied as for the *WHERE* clause.

6.3.3.3.4 Analyze and Rewrite Correlation Clause

The access of correlations in SARI-SQL is defined by the *OVERCORR* clause and allows to define pre-selector over events by restricting the space of available event object types in the *FROM* clause. The *OVERCORR* clause is applied after the *FROM* clause and holds a list of correlation name items. In case that more then one correlation is used aliasing is applied in order to be able to define which event object type in the *FROM* clause belongs to which correlation to avoid ambiguity. In case that a *FROM* item is not aliased, but aliasing is used for correlations, then all event objects of the corresponding types are selected. The internal representation and organization of correlations in the Event-Base is described in detail in Section 5.5 and is the underlying data structure for SARI-SQL.

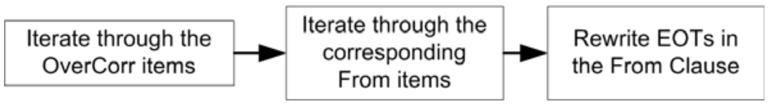

Iterate through the OverCorr items
Iterate through the corresponding From items
Rewrite EOTs in the From Clause

Figure 6.46: Correlation Clause Analysis and Rewriting Process

The planning process of the correlation clause evaluation is illustrated in Figure 6.46. The process itself is again quite simply structured. For every correlation item the corresponding referenced items are walked through and a rewriting strategy is applied. The most complex part of this process is to perform the rewriting for correlations and their corresponding *FROM* items according to the described correlation management in Section 5.5.

To give an impression of the resulting rewriting (b) the reader should consider following simple SARI-SQL query example (a) containing a correlation access, where the target RDBMS of the Event-Base is the MS SQL Server 2005. This SARI-SQL query example depends on the example shown Figure 6.14 that was introduced at the beginning of this chapter in Section 6.3.1.

Example:

a) *SELECT start.StartLocation, end.EndLocation*
FROM TransportStart start, TransportEnd end
OVERCORR TransportInfo

b) *SELECT start.TransportStart, end.AttrInt2 FROM*
(SELECT ETgeneratedId0., corr.**
FROM EBCorrelations corr, EBCorrelations2Events corr2event,
dbo.[TransportStart_X] ETgeneratedId0
WHERE corr.CorrelationSetId = 'TransportInfo'
AND corr2event.CorrelationGuid = corr.CorrelationGuid
AND corr2event.eventguid = ETgeneratedId0._guid) start
FULL OUTER JOIN
(SELECT ETgeneratedId1., corr.**
FROM EBCorrelations corr, EBCorrelations2Events corr2event,
dbo.[TransportEnd_X] ETgeneratedId1
WHERE corr.CorrelationSetId = 'TransportInfo'
AND corr2event.CorrelationGuid = corr.CorrelationGuid
AND corr2event.eventguid = ETgeneratedId1._guid) end
ON start.CorrelationGuid = end.CorrelationGuid

6.3.3.4 Execution, Evaluation and Rendering

The *Query Evaluation* is the last SARI-SQL query evaluation step. The previous block used an optimized internal SARI-SQL query model for query planning which consists of several rewriting strategies that are heavily based on previously applied model analysis procedures. The result of the *Query Planning* block is an SQL query statement compliant to the Event-Base repository schema, described in Section 5.5, and taking into account the target RDBMS systems.

The *Query Evaluation* block is now responsible of executing the prepared and pre-processed SARI-SQL query. This process block basically consists of an execution phase and a post-evaluation phase of the results, which is referred as *Evaluation/Rendering* in Figure 6.22. The execution phase itself is just executes the rewritten query statement, previously created in the *Query Planning* block, against the Event-Base. The result of this execution is a table with columns according to the rewritten projection items. As the analysis and rewriting strategies created a result overhead for handling not representable items such as nested types it is necessary to post-process the result table. So for instance the SARI-SQL *WHERE* and *ON* clauses broke down OR, AND and EQUALS operator nodes, in order to span up decomposition points for a sub-expression dissemination. Parts of those the expressions could be directly pushed onto the Event-Base and parts of the conditions and joins must be post-evaluated in this phase in memory with the help of EAExpressions.

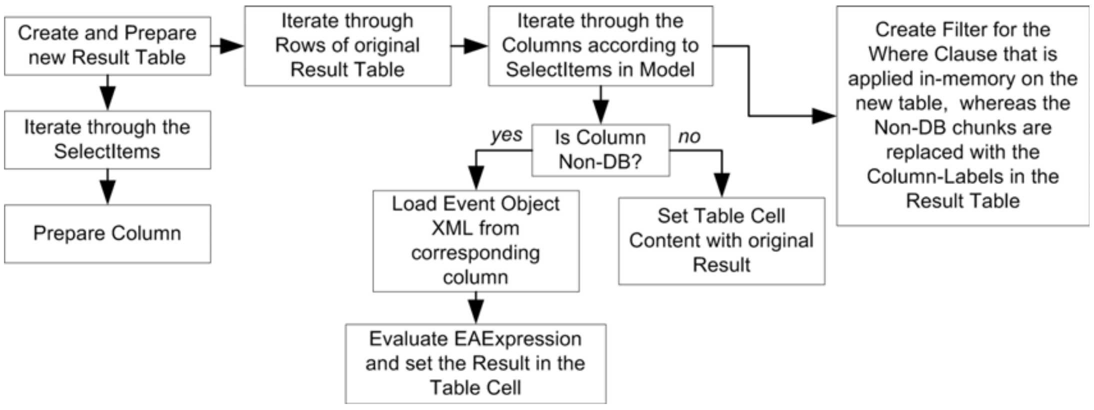

Create and Prepare new Result Table
Iterate through the SelectItems
Prepare Column
Iterate through Rows of original Result Table
Iterate through the Columns according to SelectItems in Model
Is Column Non-DB?
yes
no
Load Event Object XML from corresponding column
Evaluate EAExpression and set the Result in the Table Cell
Set Table Cell Content with original Result
Create Filter for the Where Clause that is applied in-memory on the new table, whereas the Non-DB chunks are replaced with the Column-Labels in the Result Table

Figure 6.47: Result Post-Evaluation Process

In the following the major process steps (shown in Figure 6.47) of the post-evaluation of the results will be described and for better understanding the example of the *WHERE* clause shown in Figure 6.42 in Section 6.3.3.3 will be extended for explanatory purposes.

The post-evaluation process itself is based on iterating through the row of the returned result table. But before the iteration of the rows is conducted, a new and clear result table is created in memory. The columns of that newly created table are determined by the items *SELECT* clause part of the SARI-SQL model, which also holds the overhead items such as on the Event-Base unrepresentable nested types. Such an item is for instance the item *TransportInfo.Destination* shown in the example Figure 6.48. Bear in mind that the original result table does not contain these type of overhead columns anymore as they have been rewritten to retrieve the whole *EventObjectTypeXml* therefore they have to be recreated.

The next major step in the process is to walk through every *SELECT* clause item (i.e. the columns in the newly created table) and check whether it is a non-DB type or not - just like in the previous analysis step of the *Query Planning* block. If it is a non-DB type the corresponding *BelongTo* XML representation of the event object is loaded from cell of the original result table. The loaded XML event object is then used to evaluate the EAExpression of the SelectItem. The result is then written away into the corresponding result table cell. Otherwise, if it is a DB type, the content of the corresponding orginial table cell is written to the new result table cell.

At the end of the process, a table is set up containing all data that is required to post-evaluate conditional items and joins that are not directly representable by the Event-Base. For that purpose the previously not evaluated parts of the *WHERE* clause are rewritten to match the column labels/names of the new evaluated result table. By doing that it is possible to apply an in memory filter or to create joins over such non-DB type items.

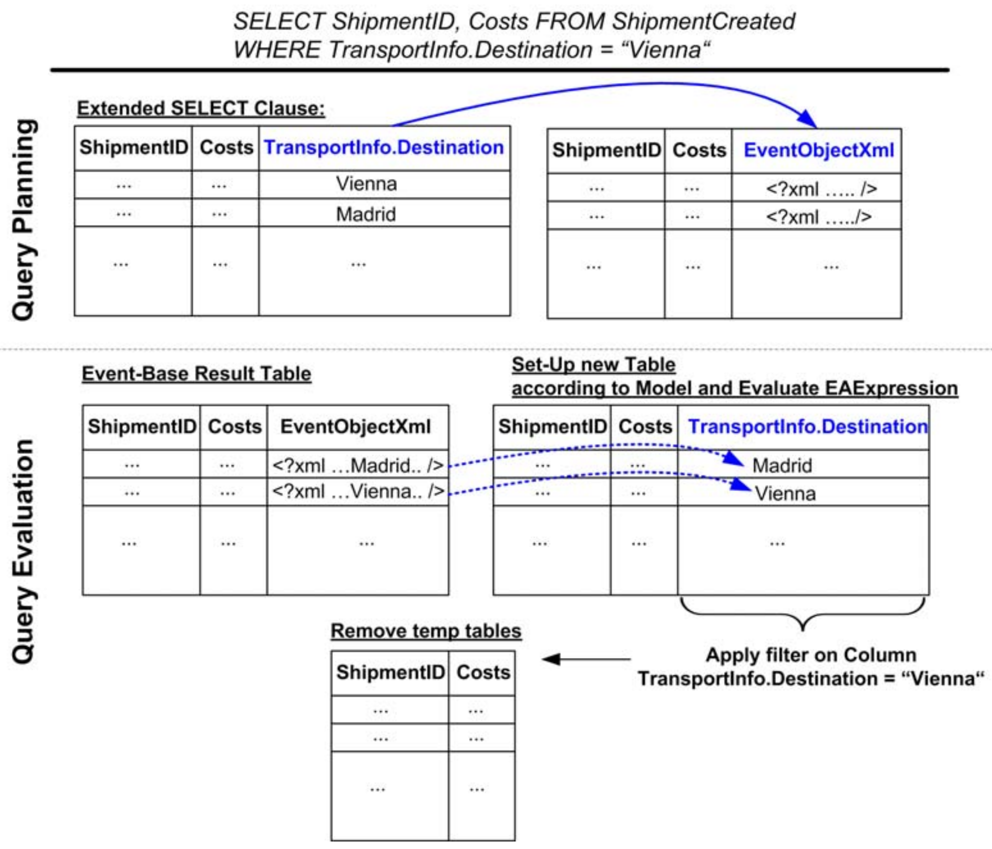

SELECT ShipmentID, Costs FROM ShipmentCreated
WHERE TransportInfo.Destination = "Vienna"
Query Planning
Extended SELECT Clause:
ShipmentID
Costs
TransportInfo.Destination
Vienna
Madrid
EventObjectXml
<?xml ..... />
Query Evaluation
Event-Base Result Table
<?xml ...Madrid.. />
<?xml ...Vienna.. />
Set-Up new Table according to Model and Evaluate EAExpression
Remove temp tables
Apply filter on Column TransportInfo.Destination = "Vienna"

Figure 6.48: Post-Evaluation Process Example

In order to ease the understanding of this process the steps are illustrated with an example in Figure 6.48. The first action was taken in the *Query Planning* block where the *TransportInfo.Destination* item of the *SELECT* clause was created as it is a nested type and thus a non-DB type. Further that item was rewritten to access the corresponding *EventObjectXML*, which is in this example the *ShipmentCreated* event object type. The result table in the *Query Evaluation* block contains the *ShipmentID, Costs* and the *EventObjectXml* holding the event object instances wrapped into a XML structure. Now the new result table is created holding the original *TransportInfo.Destination* instead of the *EventObjectXml*. During the process every row of the original result table is walked through and as the first two columns were DB types they can be taken directly into the final result table. However the third column (i.e. the *TransportInfo.Destination*) is treated as an EAExpression that is evaluated against the event object, wrapped inside XML, of the corresponding row. The result of the EAExpression evaluation process is then stored into the new result table. At the end of the process the *WHERE* clause is rewritten to match the label of the *TransportInfo.Destination* column. Then it is applied as a conditional filter against the new result table and every tuple is excluded that does not contain *"Vienna"* in the third column. Finally the overhead helper columns are stripped away, so that only the explicitly defined projection items of the SARI-SQL *SELECT* clause are left over in the result table.

6.4 Summary

This *SARI-SQL Query Language* chapter introduced the two languages SARI-SQL and EAExpression, which are an important feature for event processing systems in particular for the complex event processing system SARI [81], its processing components and the efficient up-to-date operational storage extension - the Event-Base.

SARI-SQL is a query language for retrieving near real-time events and create conjunctions with historical events, metrics and scores and is in contrast to Event Clouds indexing approach [76][73][90] a formally structured solution that extends ANSI-SQL. It creates an abstraction of the event type model by encapsulating a lot of overhead and putting a layer over events and their internal data structures. EAExpressions on the other hand pay a special attention to the nature of events and their special characteristics such as providing an expressive and easy to understand language for accessing events, both during the design phase of event processing applications and also during runtime to perform evaluations on events. The EAExpression language is a subset of the SARI-SQL language and can be used independently from SARI-SQL.

The chapter is mainly organized in a pragmatic way to explain in iterative steps the general concepts, applications, syntax, semantic and how the two languages are interwoven and play together. The syntax of both languages were described by the help of a modified Extended Backus-Naur-Form (EBNF) notation and illustrated with examples based on concrete scenarios for better understanding. In case of

EAExpressions the syntactical and semantical concepts were discussed such as how event object types, their headers, how to work with multi-value attributes such as collections and dictionaries and other miscellaneous term constructs. In case of SARI-SQL the integration of the EAExpression syntax into the main query blocks were introduced and further the usage of functionalities were described such as how to retrieve events, access correlations and how to define joins. In addition the abstract syntax tree for the language constructs were formalized and described in detail as they are an important concept for the languages evaluation processes. Furthermore syntactical and AST parts of the EAExpressions were overridden by SARI-SQL in order to be able to define constraints for steer the expressiveness. At last the validation and evaluation processes of both languages were explained. The main focus was set to the evaluation of SARI-SQL queries itself as EAExpressions can be treated from a validation and evaluation point of view as a black-box.

The main advantage of SARI-SQL is that is enhances the main building blocks of the well known ANSI-SQL syntax (*SELECT, FROM, WHERE, ...*) in order to encapsulate details of the event processing domain. Therefore SARI-SQL allows to access and reference (by aliasing) event object types through the *FROM* clause. The *SELECT* clause allows user to directly access event objects types and their attributes without knowing anything about their internal representation. Furthermore it provides a simple access to the different attributes types such as header attributes, collection items and dictionaries. In a *WHERE* clause it is possible to apply conditions in order to reduce the set of resulting tuples. Comparable to ANSI-SQL it is also possible to define joins (also left, right and full outer joins) between attributes of event object types. The language also ships a rich set of pre-implemented functions containing various mathematical, statistical and other common functions such as type converters or list operators. One of the most important features of SARI-SQL is the possibility to define specific correlations within a query by using the *OVERCORR* clause. A correlation allows to retrieve events that share a certain relationship. This enables to track down causal relationships for instance, detect patterns and perform event mining tasks. At last SARI-SQL also allows to access metrics and scores maintained in the Event-Base.

Nevertheless the chapter provided a comprehensive introduction, of the main contribution of this dissertation, the SARI-SQL query language and its related concepts and constraints. To set the contribution of this chapter into relation and context with other work, in the field of event processing languages, a comparison of several important aspects is introduced.

7 Related Work and Comparison

7.1 SIENA

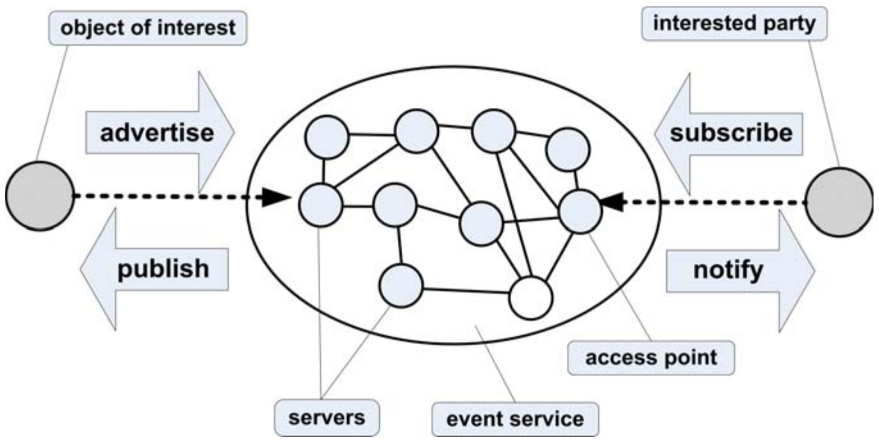


Figure 7.1: SIENA Distributed Event Notification Service [23]

SIENA (Scalable Internet Event Notification Architecture) [23] is a multi-broker event notification service (shown in Figure 7.1), based on the publish/subscribe paradigm, focusing on maximizing the expressiveness and maintaining the scalability in wide-area networks.

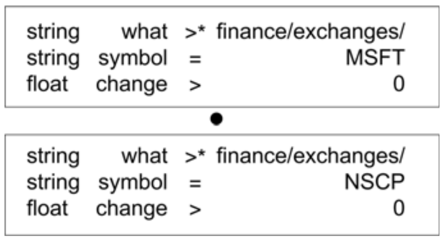
The main problem instance in such wide-area networks is the big number of interacting components that goes hand in hand with a potentially high number of notifications between those interacting communication partners.

SIENA (shown in Figure 7.1) provides a pub/sub notification interface towards the interacting clients and works as a kind of access point where the publishing, or in terms of event-based interaction, the producing components can use them to place advertisements of future event notifications. On the other side of the communication, the consuming components, can issue their interest in event notification through specifying filters.

An advertisement is broadcasted over the whole broker network and it is used to prepare the routing of event notification in such a network. A subscription is issued internally to the brokers by going back the previously defined path of the advertisement. The advertisements and subscriptions create routing entries that are used for event notifications to match the subscription and their routing direction. This also brings a drawback with it a the advertisements have to be processed by all brokers in the network. In large networks this may lead to unpredictable delays until the advertisements have been processed.

An important beneficial aspect with the subscription distribution towards the notification producer is that event notification can be filtered out early. This early filtering could save network bandwidth as events, in which no consumers are interested in, can be filtered out immediately.

The underlying data model is a notable asset of SIENA as the data model consists of typed attributes. Each attribute consists of a tuple containing the attributes *type*, *name* and *value*. Attributes of a notifications belong to a set of predefined types that are available in common programming languages. This event typing has be chosen in order to preserve scalability in wide-area networks. If the event is an explicit notification type a global authority would be required to manage those types.



```
string   what  >* finance/exchanges/
string  symbol  =               MSFT
float   change  >                  0
                ●
string   what  >* finance/exchanges/
string  symbol  =               NSCP
float   change  >                  0
```

Figure 7.2: Example of an event pattern in SIENA [23]

SIENA allows to define filters and patterns for retrieving event notifications (e.g. defining subscriptions).

- **Filters** in SIENA are matched against the attributes of one event notification. They are basically simple constraints on attributes with operators for equality, ordering relations, strings and for defining wildcards.
- **Patterns** are executed against one or more event notifications which allows to define patterns that can correlated events. However the syntax of the pattern language is limited to the definition of a list of filters on events that are temporarily ordered. Figure 7.2 shows a pattern example taken from [23]. This pattern

detects a price increase of the stock MSFT followed by an increased of the stock NSCP.

7.2 Gryphon

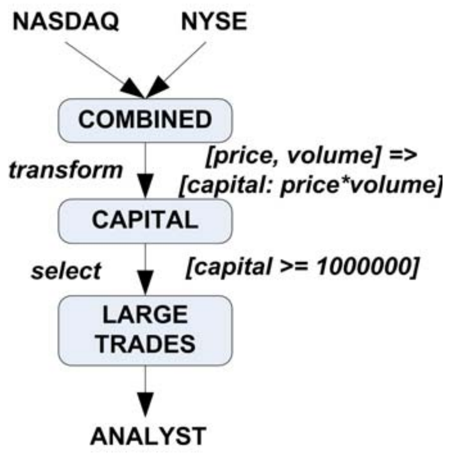


Figure 7.3: Information Flow in Gryphon [84]

Gryphon [51] is a project at the IBM Research IBM T. J. Watson Research Center and is a content-based publish/subscribe notification system. Interacting components can attach themselves to the system by making use if the Java Message Service (JMS) interface API. A special focus was set on creating a redundant broker network for routing notifications with the ability of providing guaranteed delivery while preserving scalability and availability.

Gryphon is described by the authors of [84] as an augmentation of the publish/subscribe with following features:

- **Content-based subscription** which means that the consumer is capable of defining subscriptions based on the internal data structures or attributes of the messages itself.
- **Event transformations** that allow to manipulate events.
- **Event stream interpretation** which allows to collapse and expand back sequences of events.
- **Reflection** that provides meta-events to manage the system.

The Gryphon system is based on information flow graphs (acyclic directed graphs) which are used to define the exchange of event notifications between the producers and consumers. Each of the nodes in the graph can contain specific types of events whereas the edges are capable of selecting/filtering or transforming them.

The example in Figure 7.3 shows the two different stock tickers NASDAQ and NYSE. Those two event producers are combined in the information flow graphs to one event notification stream. The first edge transforms the the event notifications, while the second one is applying a filter to the event notifications that selects out events with a capital less than 1.000.000,–.

The nodes in such a graph are partitioned into several brokers, that don't necessarily have to be physical brokers. There are three types of brokers in Gryphon. The first one is a a broker that is connected to event producers - e.g. components that are publishing event notifications. The second type is connected to consuming components - e.g. components that are subscribed to event notifications and the last type of brokers are intermediate brokers. The mapping is defined before deployment which makes changes and adoptions during runtime difficult. However, as the physical implementation of the flow graphs is open, it is possible to alter the flow of patterns and transformations.

7.3 JEDI

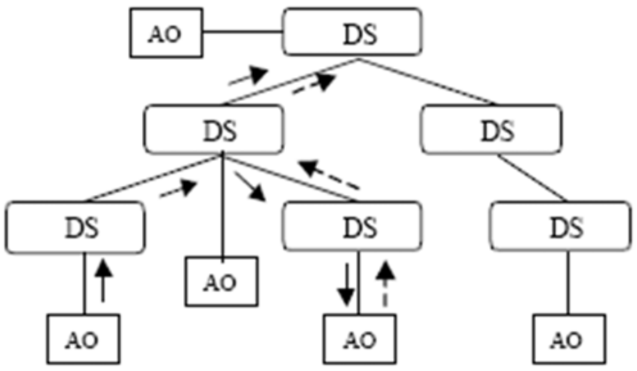


Figure 7.4: JEDI's Hierarchical Strategy [34]

JEDI (Java Event-based Distributed Infrastructure) [33] is a distributed communication middleware based on the publish/subscribe communication paradigm. The main characteristics of JEDI is that it is based on multicast, nor the destination or the producing source is defined, the notification delivery is guaranteed and mobility is ensured through the concept of reactive objects [34].

The event notification routing is carried out through so called event dispatchers, which is physically either organized as a central component or by multiple distributed components that are then interconnected through a hierarchical topology. Subscriptions are forwarded from the issuing nodes to the top root node. The dispatching servers store the subscriptions in internal data tables. If a notification has been received by a dispatching server it forwards it to components that provided the dispatcher with a matching subscriptions. This forwarding is executed until the consumer component, that published a subscription, have received their event notification. In contrast to SIENA, advertisement of notifications are not used by JEDI.

JEDI introduces the notion of active objects (AO) which is a component that is producing or consuming events. Figure 7.4 shows an illustration of the event notification distribution strategy where an AO on the top is producing event notifications and the event dispatchers are distributing them down to the subscribed AO consuming components.

The data model of JEDI is built upon key-value pairs where each of event notification consists of a name and attributes. Subscriptions are called event profiles in JEDI and contain filters also based on key value pairs. Lets consider an event *TransportStart* with the attributes *DepartureLocation*, *DateTime* and *Product*.

A subscriptions in JEDI would contain following collection of tuples: *<EventName, TransportStart>; <DepartureLocation, *>; <DateTime, *>, <Product, Tonsalumn>*. In this case all *TransportStart* event notifications would be filtered out that contains the product *Tonsalumn* regardless of the content of the *DepartureLocation* or *DateTime* attributes.

JEDI supports mobility of components and their temporal disconnection. If a component is disconnected from the distribution network all subscribed notifications are stored. If the component is later online again the notifications will be flushed to the previously absent component.

7.4 HERMES

Pietzuch and Bacon introduced Hermes [71] [8] that considers itself a distributed event-based middleware. The system is aiming at covering traditional middleware functions like type-checking of invocations, reliability, access control and transac-

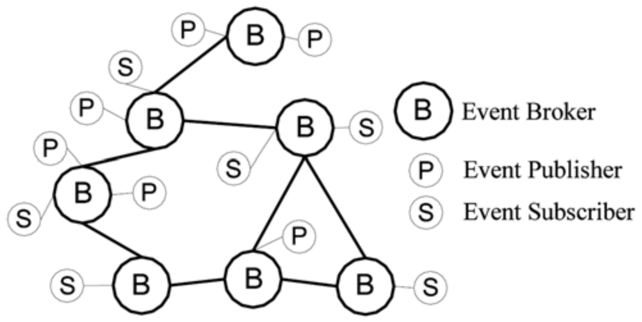


Figure 7.5: Application Built in Hermes [8]

tions. Hermes is based on the publish/subscribe paradigm making use of type- and attribute-based subscription mechanisms. Furthermore, Hermes is taking traditional middleware features like interoperability, reliability and usability into account.

Figure 7.5 shows an illustration of a Hermes application. The two main components are so-called event clients and event brokers. The event clients are the producing and consuming components in this model and thus they exchange event notifications through subscriptions and publications. The event brokers provide the necessary middleware functionalities. The event clients can connect themselves to the brokers and issue subscriptions or send event notifications. The broker nodes are interconnected with each other and they deliver the published event notifications to the corresponding subscribed consumers.

The event brokers form a logical routing network for event notifications. Advertisements are used to construct an event dissemination tree for between publishers and subscribers of event notifications. Hermes introduces so called rendezvous nodes as a meeting point for advertisements and subscriptions. There exists one of these special rendezvous nodes for each event type in the network and each of the nodes is known to publishing and subscribing components. These nodes are replicated throughout the network to ensure fail-over scenarios. Event subscriptions are distributed through the network in reverse order of the way that the advertisements took. Each broker that is passed stores the subscription (e.g. filter) internally and applies it whenever an event notification passes by.

In contrast to other publish/subscribe systems Hermes is using type-based approach for event notifications instead of simple key-value pairs. Every event in the system corresponds a specific event type that helps to secure robustness of the system as type-safety can be ensured during runtime. The event types can be defined in a hierarchical inheritance structure like known in object-oriented programming languages. Subscriptions can then be applied to parent types which will also take inherited children into account.

The subscription language itself is a message that selects an event type and defines a attribute filters on the specified event type. The filter is an XPath expression that is evaluated against incoming event notifications. Such a subscription including an attribute filter is shown in Figure 7.6.

```
<?xml version="1.0" encoding="UTF-8"?>
<h:hermes xmlns:h="http://www.cl.cam.ac.uk/opera/hermes">
  <subscription>
    <subscribe typename="LocationEvent">
      <typeattr>
        <xpath>child::*[child::id>3141 and child::position="FE02"]</xpath>
      </typeattr>
    </subscribe>
  </subscription>
</h:hermes>
```

Figure 7.6: Hermes Subscription Message Definition [71]

7.5 REBECA

REBECA [42] is a distributed publish/subscribe event notification system especially aiming at large-scale e-commerce applications. The main design effort of the architecture was laid on system scalability and the optimization and evaluation of notification routing algorithms. In REBECA new routing algorithms can be added in terms of advertisement and subscription forwarding in the overlay network. The interface towards the client components is decoupled from the underlying routing algorithm.

REBECA is relying on acyclic graphs representing the overlay network of brokers partly comparable to SIENA. To preserve the scalability and fail-over scenarios of such a network, redundancies can be applied of components can be applied. The constituents of the network are shown in Figure 7.7. The network is built up by three different types of components:

- **Local Broker:** The local broker acts as an access point and is usually part of an application library. Such a broker is connected to a border broker and is not considered as a regular part of the broker network
- **Border Broker:** The border brokers form the logical borders of the overlay network and are directly connected to the clients.

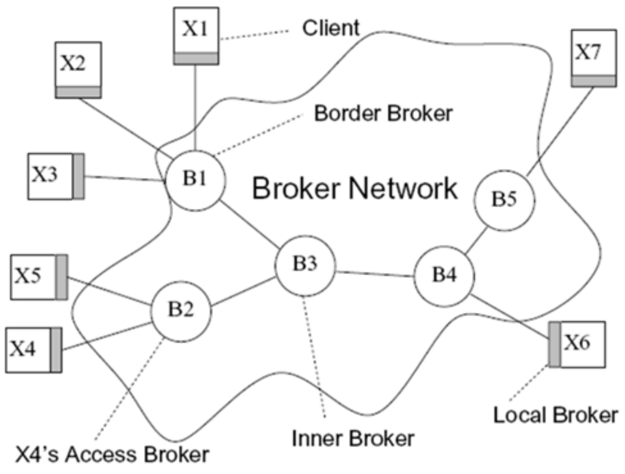


Figure 7.7: The Router Network of REBECA [93]

- **Inner Broker:** The inner brokers form the inner circle of the system and have no outside (e.g. client) connections.

Subscriptions are propagated through the network of brokers and forming a delivery path through routing tables. Those routing tables contain a filter and link pair, where the link contains the reference to the source of the subscription. In the simples routing case all brokers maintain a global knowledge about active subscription which has the advantage that it can reduce the network traffic as it pushes the filters towards the notification producing components (comparable to SIENA). However it also creates a huge amount of overhead as large routing tables have to be maintained on every node. In addition there is a concept subscription merging introduced [64] to reduce the amount of states kept on brokers.

The data model of an event notification in REBECA consists of key-value pairs representing attributes. The attributes can be of simple runtime types like (int, string, bool, ...). Subscriptions can be defined through creating filters that set constraints on attributes. A filter can contain several attribute filters where one attribute filter is constraining a specific attribute of a notification. Such an attribute constraint can apply operators like equality, inequality and comparison operators.

7.6 Aurora

The Aurora project [1][92] is an event stream processing engine designed and developed at the Brown University together in cooperation with the MIT and

Michael Stonebreaker. The systems considers itself as a high performance stream processing engine that ships with a set of extensible stream operators and a workflow oriented processing approach including a GUI tool for manipulating workflow diagrams.

The fundamental idea behind the development of Aurora was to encounter the flaws of traditional RDBMS systems, that follow the HADP (human-active, DBMS-passive)[1] model which is not able to cope with the requirements of monitoring applications. Monitoring applications need to process large amounts of data streams, with imprecise data (for instance coming from sensors) and computation latency is an important requirement (e.g. real-time processing).

Aurora aims at integrating multiple streams coming from data sources into a graphical modeled workflow graph. The workflow system is a directed and loop-free graph containing operators where the processing steps are represented through arrows and boxes. Further Aurora is providing a features that allows to maintain and access historical data and perform ad-hoc queries.

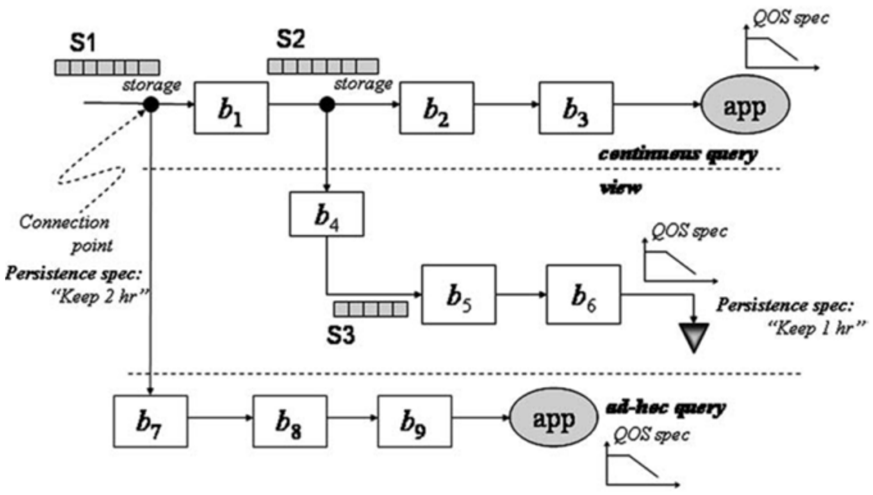


Figure 7.8: The Aurora Query Model [1]

Figure 7.8 shows a conceptual illustration of Auroras processing workflow and how continuous queries can be applied on incoming data streams. On the top left side of the figure is the entrance point for data streams. The boxes represent operators that can be applied to the data streams. Data that has been processed through the graph is usually gone. There are black dots on the map that work as a persistence container and are capable of storing data for a specific amount of time. The persisted historical data can be retrieved by placing ad-hoc queries against the containers. Paths without connected applications (shown in the middle of the figure) can also persist data for later processing. So for instance if an application is attached later in time to the processing graph it is still capable of absorbing events from the processing network.

Auroras stream data model is represented by a sequence of tuples, containing attribute-value pairs, where each tuple is marked by a timestamps that defines the entrance time of the event in the aurora network. The query language SQuAl (Stream Query Algebra) contains seven operators that are comparable to operators known from the relational databases with regards to the requirements of stream processing.

There are two groups of operators in in Aurora. The first one are order-agnostic operators and the second one are order-sensitive operators. This is because Aurora is not guaranteeing the delivery order, which is an advantage when processing tuples by priority.

- **Order-agnostic operators**
  - **Filter:** Matches incoming tuples against predicates. In case they satisfy them the tuples is diverted to a specified output stream.
  - **Map:** Is a projection operator that basically transforms input tuples.
  - **Union:** Is used to merge several input streams into one output stream.
- **Order-sensitive operators**
  - **BSort:** Is an operator that sorts an input stream based on buffering the items as it is hard to sort an infinite stream of data.
  - **Aggregate:** Defines a sliding window over data streams to calculate for instance the average price of stock ticks in the last hour.
  - **Join:** This operator is comparable to inner joins known from relation databases. It allows to select attributes from two streams and let them join.

7.7 Medusa

Medusa is stream processing system basically using Auroras query capabilities [28][10][92] with the goal to enable the distribution of evaluation of Aurora queries. Medusa if focusing, contrast to the Aurora project, on developing a distributed infrastructure in order to create a loosely coupled network of stream processing components.

According to [10] the major benefits and improvements of Medusa are that is capable of scaling up the stream processing on multiple distributed nodes, providing fail-over handling and securing a high-availability, providing services to integrate a wide set of source streams and handling high peaks of load. This is achieved by setting up an overlay network of distributed nodes.

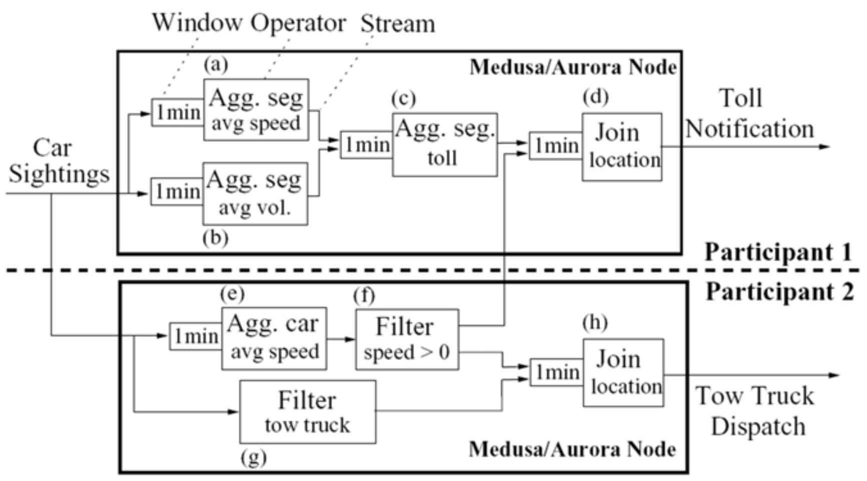


Figure 7.9: Example of a distributed Medusa Query [10]

Figure 7.9 shows an example from the report [10] how event streams are processed with Medusa making use of Aurora. In this example the input stream of "car sightings" and splits them into "toll notifications" and "tow truck dispatch". The first boxes above are calculating the average speed and the average volume of the traffic. Taking those two results are used to calculate the toll of the segment. The result is then joined with the lower participant that is filtering out tow trucks and cars that are not moving (e.g. maybe broken down).

7.8 Borealis

Borealis [2] is a distributed stream processing engine build upon Aurora [1][92] and can be seen as its successor. The project addressed several requirement for the development of the new stream processing engine which were driven by insights and experiences gained from previous projects.

The project explicitly addresses [2] the problem of dynamic revision of query results which sometimes necessary to correct errors in previously received data. Further Borealis is aiming at providing support for query modifications during runtime with a low overhead, fast and automatic modifications and providing high scalability with growing demands in heterogeneous environments including optimization strategies and fault tolerance.

Borealis data model was extend in contrast to Aurora with the ability to support the revision of data from incoming streams. Therefore there are now three types of message available. An insertion message (which are tuples known from Aurora), Deletion messages and replacement messages. The processing graphs from Aurora have been reused and are explained in Section 7.6.

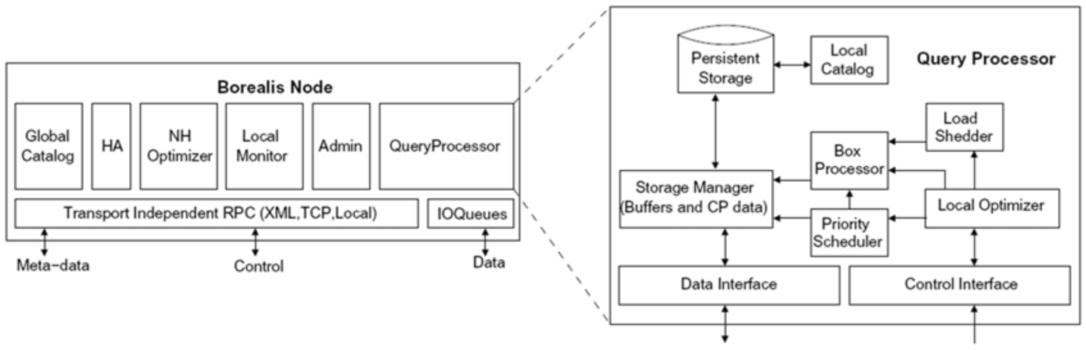


Figure 7.10: Borealis Architecture [2]

Figure 7.10 shows the Borealis architecture explained in detail in the article [2]. The most interesting constituents in context of this thesis is the Query Processor which is the execution place for queries and contains several other supporting components. So for instance it holds a local optimizer that is handling performance improvements during runtime by communicating with other components ans so for instance is capable of prioritizing events in case of peak load. Basically the query engine from a conceptual point of view is comparable to Aurora. There is also a storage manager in place that is persisting data streams that are required during processing steps in the graph known from Aurora.

7.9 SASE

SASE considers itself as a complex event processing systems which was recently introduced in the article [48] with a RFID application scenario for event stream processing. SASE offers an expressive and user-friendly continuous query language that allows to define sequence patterns which takes temporal order of events into account. Further it supports sliding time windows that can be spanned over long durations (hours or days) and allows to retrieve intermediate result sets. Further it is capable of persisting event processing results and perform combinations of continuous query results with database queries.

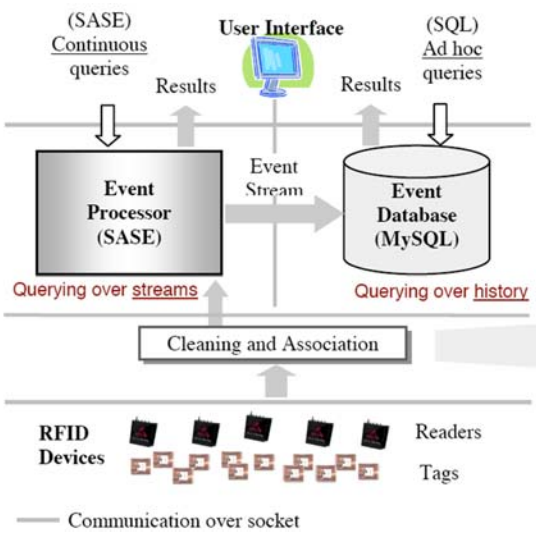
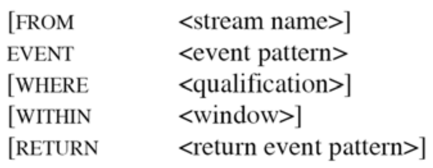


Figure 7.11: SASE Architecture [48]

The Figure 7.11 shows the overall SASE architecture within the RFID application scenario. The lowest layer is the event producer that is reading the RFID tags. The second layer performs data cleansing like removing duplicates, adding timestamps and generating the event according to a predefined schema. These events are then pushed into SASEs CEP engine where they are running against continuous queries. Filtered events can be transformed by rules to archiving purposes. At the top there is a user interface that allows to define continuous queries and let them place ad-hoc queries against the data repository.



```
[FROM      <stream name>]
EVENT      <event pattern>
[WHERE     <qualification>]
[WITHIN    <window>]
[RETURN    <return event pattern>]
```

Figure 7.12: SASE Query Syntax[48]

The continuous query language described in detail in [48] is on the first glance comparable to ANSI-SQL. The Figure 7.12 shows the query syntax of SASEs language. The FROM clause defines the input event streams that should be processed while the EVENT clause defines the event-pattern that is matched against the stream. The WHERE clause allows to impose predicates on attributes of events. That allows some kind of inner join constructs. The WITHIN defines the length of the sliding time window in which the patterns are active. The last clause is the RETURN which transforms the streams for output.

7.10 Esper

Esper [30][16] is an Open Source event stream processing solution for analyzing event streams. Esper supports conditional triggers on event patterns, event correlations and SQL queries for event streams. It has a lightweight processing engine and is currently available under GPL licence.

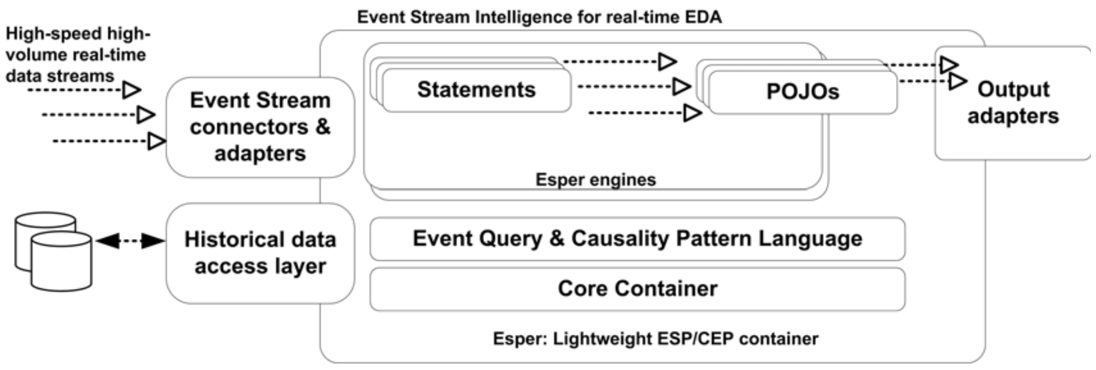

Event Stream Intelligence for real-time EDA
High-speed high-volume real-time data streams
Event Stream connectors & adapters
Statements
POJOs
Output adapters
Esper engines
Historical data access layer
Event Query & Causality Pattern Language
Core Container
Esper: Lightweight ESP/CEP container

Figure 7.13: ESPER Architecture[31]

Figure 7.13 shows the high level architecture of Esper consisting of event source adapters that can be attached to the event producing components and output adapters which allow to write away data to custom data sources for instance. Further it contains a data access layer that encapsulates the historical data repository.

Esper is a lightweight event processing engine that represents its events through simple POJOs (Plain Old Java Objects) whereas the attributes and their types are determined through the Java class attributes. It supports also nested type structures through dictionaries or collections and supports inheritance structures that are considered by the query statements. Esper furthermore requires an event type definition in XML and thus supports validation through XML. It is possible to access events through the event query language, through XPath or programmatically through Java code.

Espers query language EQL is comparable to ANSI-SQL but is extended with concepts to conform the requirements of event processing. Such features are for instance moving time windows and the support of aggregations. Figure 7.14 shows the EQL syntax declaration which is comparable to SASEs introduced in the previous section.

The SELECT clause is a projection of an event type that is resulting from the query. The FROM clause in the statement defines one or more event streams that are under evaluation. The WHERE clause a filter with a set of comparator operators that can be applied on event attributes. Further it allows to create joins, similar to inner join, upon events. The OUTPUT clause is used to control the output rate of events in intervals. Time Windows are defined in the FROM clause upon selected event streams. A complete documentation of the queries abilities can be found in the Esper documentation [30].

```
SELECT select_list
FROM stream_def [as name] [, stream_def [as name]] [,...]
[WHERE search_conditions]
[OUTPUT output_specification]
```

Figure 7.14: Simple Esper Query Example

7.11 Amit

Amit (active middleware technology) [3] is an event stream engine whose goal is to provide high-performance situation detection mechanisms. The intention of the development of Amit was to close the gap between the identification of events and the detection of situations that require a corresponding reaction. Amit is both a development framework and an application that allows the users and developers to build fast and relatively easy situation detection and reaction applications. For these purposes Amit offers a sophisticated user interface for modeling business situations based on the following four types of entities: events, situations, lifespans and keys.

Figure 7.15 shows the high-level architecture of Amit. Amit distincts two types and two source classes of events. The “concrete event” is an event that occurs in the real-world and reflects a significant state change that has been observed. An “inferred event” on the other hand is more or less and artificially created event, based the occurrence of concrete events and the conclusions drawn out of those events. An event is the base entity and can be specified with a set of typed attributes. Events

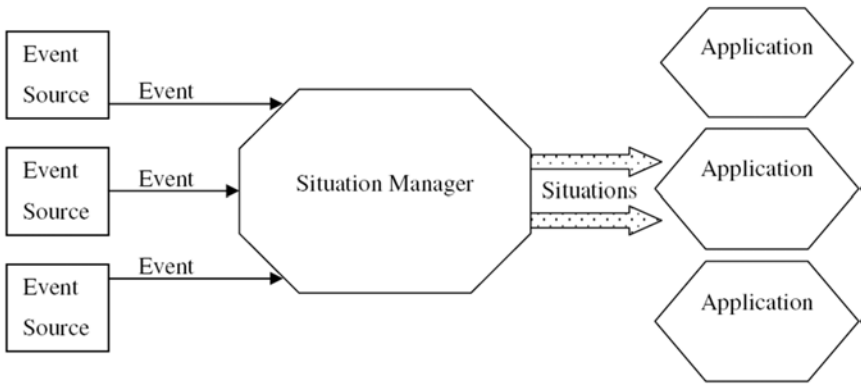


Figure 7.15: Amit Situation Manager's High Level Architecture [3]

are represented through event instances which can share relationships among each other. Every event corresponds to an event type that defines the schema of attributes. "External events" are event classes that in Amit that have been received by the situation manager from external sources. "Internal events" are events are fired upon detected situations by the situation manager. The concept of lifespans allow the definition of time intervals wherein specific situations can be detected. A situation is the main instance for specifying queries. Detected situations are signalized by propagating internal events.

Amits situation detection is split up into three phases. The first phase collects event instances during runtime. In the second phase specific event instances are selected that could play a role in the situation detection and the last step events that play a role in the situation are removed from the collection. The situation detection itself can be defined through the usage of a number of operators. Amit supports quantifier attributes, join operators, counter functions, temporal operators and also event absence operators that can be applied within a define lifespan.

7.12 Summary and Comparison

In the following section a categorized overview of the solutions from this chapter is presents. A fine grained comparison and classification of the various solutions and research projects is not possible due to several reasons described in the Sections 2 and 3. For instance the development of event-based systems can be originated to the fields of append-only databases, zero/active data warehousing, the publish/subscribe paradigm and were strongly influenced by middleware concepts. During time, different approaches for processing events forked apart - each approach addressing specific issues such as an expressive subscription language, scalability of event processing systems or a sophisticated event processing model. Therefore the table presents a map that allows the reader to allocate the solutions according to main concepts.

The meaning of the categorization elements will be explained in the preceding paragraph.

- **Pub/Sub Model:** This item determines if the solution is relying on pub/sub paradigm (compare with Section 3.1.6).
- **Wide-Scale Processing:** Wide-Scale processing is referring to solutions that yaw for scalability, high-availability, fault tolerance and efficient event notification distribution.
- **Multi-Broker Model:** Solutions that are focusing on wide-scale event processing try to maximize the expressiveness and the scalability in wide-area networks. The enablers of such solutions are broker networks that provide the infrastructure for optimal notification routing and processing.
- **Attribute-Based Filter Language:** Basic filter language with simple expressiveness - mostly focusing on filtering events.
- **Attribute-Based Pattern Language:** More advanced filter language that allows to express patterns or complex filters - mostly focusing on filtering events.
- **Type-Based Filter Language:** Support of sophisticated event notification filters (compare with Section 3.1.6).
- **Content-Based Subscription:** Support of sophisticated event notification filters (compare with Section 3.1.6).
- **Event Transformation:** Language allows to transform or manipulate event notifications. Usually a rudimentary event processing feature.
- **CEP/ESP:** This item defines if the solution determines the characteristics of complex event or event stream processing solutions.
- **Workflow Oriented Processing:** The workflow oriented processing defines if the solution provides an event processing model that allows to define the process of how events are evaluated - possibly supported by a query language (compare with Section 5.1.3).
- **Rule-based Oriented Processing:** Defines if the rule processing can be performed following an event-condition-action paradigm.

- **Event Data Management:** Event data management determines that the solution is capable of maintaining and persisting events for further use such as for ad hoc queries, for enrichment or for later analysis and mining purposes.
- **Event Stream Joins:** Determines if it is possible to merge several event producing sources (i.e. streams of events) for processing.
- **Event Stream Aggregation:** Determines if it is possible to define event aggregations.
- **Event Correlation:** Determines if the solution offers capabilities to define correlations over events.
- **Sliding Time-Window:** Determines if the solution requires the use of sliding time-windows for processing.
- **SQL-like Continuous Query Language:** Determines if the syntax of the event query language is comparable to ANSI-SQL

	AMIT	**Borealis**	**Esper**	**SIENA**	**Gryphon**	**JEDI**	**HERMES**	**REBECA**	**SASE**	**SARI**
Pub/Sub Model				X	X	X	X	X	X	
Wide-Scale Processing		X		X	X	X	X	X	X	
Multi-Broker Model				X	X		X	X	X	
Attribute-Based Filter Language				X	X	X	X	X	X	
Attribute-Based Pattern Lang.				X						
Type-Based Filter Lang.				X			X			
Content-Based Subscription				X	X					
Event Transformation					X X					
CEP/ESP		X	X					X		X
Rule-based Oriented Processing										X
Event Data Management	X	X	X							X
Event Stream Joins	X	X	X							X
Event Stream Aggregation	X	X	X							X
Event Correl.	X	X	X							X
Sliding Time-Window	X	X	X							X
SQL-like CQL			X							X

8 Conclusion

This dissertation presents a comprehensive and detailed discussion of the fundamentals on event-based systems, their processing concepts and models. In contrast to event-based solutions, especially solutions originating in the field of pub/sub middleware paradigms, this component model is a generic proposal and independent of an underlying communication infrastructure. SARI is a solution that aims at providing event processing capabilities, that are capable of processing large amounts of events, providing facilities with the capability to monitor, steer and optimize business processes in real-time. The Event-Base, extends SARIŠs event processing model with an efficient up-to-date operational storage, together with retrieval mechanisms for business events for analytical as well as operational purposes. The event processing solution SARI and its extension the Event-Base are introduced to illustrate event processing solutions that follow the paradigms and best practices of event-based system. In addition, this thesis introduces the domain specific language SARI-SQL and its sub-language EAExpressions that address the problem domain of event-based and in particular complex event processing systems. The query language offers an expressive way for retrieving near real-time events and creating conjunctions between historical events, metrics and scores. SARI-SQL is a formally structured solution that extends ANSI-SQL and is allocated in the group of domain-specific languages.

The summarization of the main contributions is presented in the following:

Event-Based Component Model. In this dissertation an event component model is introduced that puts event-based systems into a broader context of event processing, defines the boundaries of such systems and the scope of event-driven components. This model decouples event-based system completely from the communication infrastructure, which offers the advantage that the capabilities of the event processing realm are not constrained by the underlying communication technology. In addition this component model allows to attach several and different types of communication infrastructures and this allows to consume and process events from a wide range of event sources.

Several Event Processing Models and Concepts of the solutions SARI and the Event-Base. One of the main contributions introduced, is the event processing system SARI and its extension the Event-Base. SARI allows to observe relevant business events to identify exceptional situations, indicates opportunities or problems combined with low latency times in decision making for supportive or counter measures. The Event-Base, on the other hand, provides an efficient up-to-date operational storage together with retrieval mechanisms for business events for analytical as well as operational purposes without the costly data staging processes known from established data warehousing solution. The design and the nature of the event models strongly constraints the capabilities of event processing query languages and thus they have a major impact on the flexibility and usability of such event-based systems. Therefore several new concepts of organizing event models are introduced such as advanced typing concepts like inheritance, exheritance and dynamic type inferencing (duck typing). A special attention was also set on event-driven sense and respond rules, which can be used to model trees with event actions within the event processing model.

SARI-SQL Query Language. A dedicated section introduced the syntax and semantics of the event query language SARI-SQL and its sub-type the EAExpressions. SARI-SQL is a domain specific language which is an important feature for event processing systems in particular for the complex event processing system SARI and the efficient up-to-date operational storage extension - the Event-Base. SARI-SQL is a query language for retrieving near real-time events and create conjunctions with historical events, metrics and scores and is in contrast to Event Clouds indexing approach a formally structured solution that extends ANSI-SQL. It creates an abstraction of the event type model by encapsulating a lot of overhead and putting a layer over events and their internal data structures. EAExpressions on the other hand pay a special attention to the nature of events and their special characteristics such as providing an expressive and easy to understand language for accessing events, both during the design phase of event processing applications and also during runtime to perform evaluations on events. The EAExpression language is a subset of the SARI-SQL language and can be used independently from SARI-SQL.

9 Appendix

9.1 EAExpression Syntax Definition

eaExpression ⇒	*boolExpr Street ZIP*
boolExpr ⇒	*conditionExpr ((AND* ∨ *OR* ∨ *XOR) boolExpr)?*
	∨ *NOT boolExpr*
conditionExpr ⇒	*isExpr ((EQUAL* ∨ *NOTEQUAL* ∨ *GTHAN* ∨
	GEQUALTHAN ∨ *LTHAN* ∨ *LEQUALTHAN)*
	conditionExpr)?
isExpr ⇒	*plusMinExpr (IS_ OP typeExpr)?*
plusMinExpr ⇒	*prodDivExpr ((PLUS* ∨ *MINUS) plusMinExpr)?*
prodDivExpr ⇒	*prodDivExprMod (STAR prodDivExpr)?*
prodDivExprMod ⇒	*prodDivExprDiv (MOD prodDivExprMod)?*
prodDivExprDiv ⇒	*specialKeywordExpr (DIV prodDivExprDiv)?*
typeExpr ⇒	*IDENTIFIER (DOT typeExpr)?*
specialKeywordExpr ⇒	*defExpr ((CONTAINS* ∨ *CONTAINSVALUE* ∨
	CONTAINSANY ∨*COLON) specialKeywordExpr)?*
defExpr ⇒	*constValue* ∨
	dotExpr ∨
	collectionExpr (LBRACK plusMinExpr RBRACK)? ∨
	LPAREN boolExpr RPAREN ∨
	MINUS defExpr ∨
	PLUS defExpr ∨
collectionExpr ⇒	*LCURLY parameterList RCURLY*
parameterList ⇒	*boolExpr (COMMA boolExpr)**
dotExpr ⇒	*eventAtom (DOT dotExpr)?* ∨
	eventExprSpecial ∨
	functionExpr
eventExpr ⇒	*eventAtom (DOT dotExpr)?* ∨
	eventExprSpecial

eventExprSpecial ⇒	*eventAtom AT IDENTIFIER* ∨
	AT IDENTIFIER ∨
	eventAtom LBRACK plusMinExpr RBRACK
	(DOT eventExpr)?
functionExpr ⇒	*functionAtom LPAREN parameterList RPAREN*
	(eventFunction)?
eventFunction ⇒	*DOT eventExpr* ∨
	AT IDENTIFIER ∨
	LBRACK plusMinExpr RBRACK (DOT eventExpr)?
constValue ⇒	*STRING_LITERAL* ∨
	CONSTANT ∨
	TRUE ∨
	FALSE ∨
	constNumericValue
constValue ⇒	*NUM_INT* ∨
	NUM_FLOAT ∨
	NUM_DOUBLE ∨
	NUM_LONG ∨
	HEX_DIGIT
eventAtom ⇒	*IDENTIFIER*
functionAtom ⇒	*IDENTIFIER*

9.2 EAExpression Abstract Syntax Tree Definition

$$
\begin{aligned}
\boldsymbol{boolExpr} \Rightarrow\ & \mathit{AND\ boolExpr\ boolExpr} \lor \\
& \mathit{OR\ boolExpr\ boolExpr} \lor \\
& \mathit{XOR\ boolExpr\ boolExpr} \lor \\
& \mathit{NOT\ boolExpr} \lor \\
& \mathit{conditionExpr} \\
\boldsymbol{conditionExpr} \Rightarrow\ & \mathit{EQUAL\ boolExpr\ boolExpr} \lor \\
& \mathit{NOT\_EQUAL\ boolExpr\ boolExpr} \lor \\
& \mathit{LTHAN\ boolExpr\ boolExpr} \lor \\
& \mathit{LEQUALTHAN\ boolExpr\ boolExpr} \lor \\
& \mathit{GTHAN\ boolExpr\ boolExpr} \lor \\
& \mathit{GEQUALTHAN\ boolExpr\ boolExpr} \lor \\
& \mathit{isExpr} \\
\boldsymbol{isExpr} \Rightarrow\ & \mathit{IS\_OP\ boolExpr\ typeExpr} \lor \\
& \mathit{plusMinExpr} \\
\boldsymbol{plusMinExpr} \Rightarrow\ & \mathit{PLUS\ boolExpr\ typeExpr} \lor \\
& \mathit{MINUS\ prodDivExpr} \lor \\
& \mathit{prodDivExpr} \\
\boldsymbol{prodDivExpr} \Rightarrow\ & \mathit{STAR\ boolExpr\ typeExpr} \lor \\
& \mathit{DIV\ boolExpr\ typeExpr} \lor \\
& \mathit{MODE\ boolExpr\ typeExpr} \lor \\
& \mathit{specialKeywordExpr} \\
\boldsymbol{specialKeywordExpr} \Rightarrow\ & \mathit{CONTAINS\ boolExpr\ typeExpr} \lor \\
& \mathit{CONTAINSVALUE\ boolExpr\ typeExpr} \lor \\
& \mathit{CONTAINSANY\ boolExpr\ typeExpr} \lor \\
& \mathit{COLON\ boolExpr\ typeExpr} \lor \\
& \mathit{defExpr} \\
\boldsymbol{typeExpr} \Rightarrow\ & \mathit{IDENTIFIER} \lor \\
& \mathit{DOT\ typeExpr\ typeExpr}
\end{aligned}
$$

eventExprSpecial ⇒	*AT dotExpr IDENTIFIER* ∨
	AT IDENTIFIER ∨
	LBRACK defExpr plusMinExpr
functionExpr ⇒	*FUNCT IDENTIFIER parameterList (functionExpEval)?*
functionExpEval ⇒	*FUNCTION_EVENT_EVAL dotExpr* ∨
	FUNCTION_EVENT_HEADER_EVAL headAttr1 ∨
	FUNCTION_COLLECTION_EVAL plusMinExpr (plusMinExpr)?
constValue ⇒	*STRING_LITERAL* ∨
	CHAR_LITERAL ∨
	CONSTANT ∨
	TRUE ∨
	FALSE ∨
	constNumericValue
constNumericValue ⇒	*NUM_INT* ∨
	NUM_FLOAT ∨
	NUM_DOUBLE ∨
	NUM_LONG ∨
	HEX_DIGIT ∨
defExpr ⇒	*constValue* ∨
	dotExpr ∨
	NEGATION defExpr
collectionExpr ⇒	*COLLECTION parameterList*
parameterList ⇒	*PARAMETER_LIST (boolExpr)**
dotExpr ⇒	*IDENTIFIER* ∨
	DOT dotExpr dotExpr ∨
	eventExprSpecial ∨
	functionExpr

9.3 SARI-SQL Syntax Definition

sariSQL	⇒	(selectExpression)
selectExpression	⇒	((SELECT selectClause) (FROM fromConstruct)
		(OVERCORR overCorrClause)?
		(WHERE whereCondition)?)
		(ORDER_BY orderByCondition)?)
		((GROUP_BY groupByCondition (HAVING havingCondition)?)?)
selectClause	⇒	selectList
selectList	⇒	((selectItem) (COMMA selectList)?)
selectItem	⇒	plusMinExpr (AS displayName)?
fromConstruct	⇒	fromList ∨ joinClause
joinClause	⇒	fromSpecForJoin INNER JOIN fromSpecForJoin
		ON onJoinClause ∨
		fromSpecForJoin FULL OUTER JOIN fromSpecForJoin
		ON onJoinClause ∨
		fromSpecForJoin LEFT OUTER JOIN fromSpecForJoin
		ON onJoinClause ∨
		fromSpecForJoin RIGHT OUTER JOIN fromSpecForJoin
		ON onJoinClause
fromSpecForJoin	⇒	(reducedDotExpr fromAlias)
onJoinClause	⇒	onJoinItem
onJoinItem	⇒	dotExpr EQUAL dotExpr ((AND ∨ OR) onJoinItem)?
fromList	⇒	fromItem (COMMA fromItem)*
fromItem	⇒	reducedDotExpr (fromAlias)? ∨
		functionExpr (fromAlias)?
whereCondition	⇒	boolExpr
overCorrClause	⇒	corrList
corrList	⇒	corrItem (COMMA corrItem)*
corrItem	⇒	corrSpec (corrAlias)?
orderByCondition	⇒	orderByList
orderByList	⇒	orderByItem (COMMA orderByItem)*
orderByItem	⇒	reducedDotExpr (ASC ∨C DESC)?

groupByCondition	⇒	*groupByList*
groupByList	⇒	*groupByItem (COMMA groupByItem)**
groupByItem	⇒	*reducedDotExpr*
boolExpr	⇒	*LPAREN selectExpression RPAREN* ∨
		conditionExpr ((AND ∨ OR ∨ XOR) boolExpr)? ∨
		NOT boolExpr
dotExpr	⇒	*STAR* ∨
		eventAtom (DOT dotExpr)? ∨
		eventExprSpecial ∨
		functionExpr
reducedDotExpr	⇒	*eventAtom (DOT reducedDotExpr)?*
havingCondition	⇒	*boolExpr*
fromSpec	⇒	*IDENTIFIER (IDENTIFIER)?*
corrSpec	⇒	*IDENTIFIER (IDENTIFIER)?*
tableName	⇒	*IDENTIFIER*
fromAlias	⇒	*IDENTIFIER*
corrAlias	⇒	*IDENTIFIER*
displayName	⇒	*STRING_LITERAL* ∨
		IDENTIFIER

9.4 SARI-SQL Abstract Syntax Tree Definition

sariSQL	⇒	*QUERY selectClause fromConstruct (overCorrClause)?*
		(whereCondition)?
		(orderByCondition)?
		(groupByCondition)?
		(havingCondition)?
selectClause	⇒	*SELECT_LIST (selectItem)**
selectItem	⇒	*SELECT_ITEM ϕ (displayName)?*
fromConstruct	⇒	*fromList* ∨
		joinClause
joinClause	⇒	*INNER_JOIN FROM_ITEM FROM_ITEM onJoinClause* ∨
		FULL_OUTER_JOIN FROM_ITEM FROM_ITEM
		fullOuterJoinClause ∨
		LEFT_OUTER_JOIN FROM_ITEM FROM_ITEM
		leftOuterJoinClause ∨
		RIGHT_OUTER_JOIN FROM_ITEM FROM_ITEM
		rightOuterJoinClause ∨
fromList	⇒	*FROM_LIST (fromItem)**
fromItem	⇒	*FROM_ITEM reducedDotExpr (fromAlias)?* ∨
		SPECIAL_FROM_ITEM ϕ (fromAlias)?
whereCondition	⇒	*WHERE_CLAUSE boolExpr*
onJoinClause	⇒	*ON_JOIN_CLAUSE boolExpr*
overCorrClause	⇒	*overCorrList*
overCorrList	⇒	*OVERCORR (overCorrItem)**
overCorrItem	⇒	*OVERCORR_ITEM corrName (corrAlias)?*
orderByCondition	⇒	*orderByList*
orderByList	⇒	*ORDERBY (orderByItem)**
orderByItem	⇒	*ORDERBY_ITEM reducedDotExpr (ASC ∨C DESC)?*
groupByCondition	⇒	*groupByList*
groupByList	⇒	*GROUPBY (groupByItem)**
groupByItem	⇒	*GROUPBY_ITEM reducedDotExpr*
havingCondition	⇒	*HAVING boolExpr*

boolExpr	$\Rightarrow$	*AND boolExpr boolExpr* $\vee$
		OR boolExpr boolExpr $\vee$
		EQUAL boolExpr boolExpr $\vee$
		ϕ
tableName	$\Rightarrow$	*IDENTIFIER*
corrName	$\Rightarrow$	*IDENTIFIER*
displayName	$\Rightarrow$	*STRING_LITERAL* $\vee$
		IDENTIFIER
fromAlias	$\Rightarrow$	*IDENTIFIER*
corrAlias	$\Rightarrow$	*IDENTIFIER*
orderByName	$\Rightarrow$	*IDENTIFIER*
reducedDotExpr	$\Rightarrow$	*eventAtom* $\vee$
		DOT eventAtom reducedDotExpr

9.5 List of Publications

Szabolcs Rozsnyai, Josef Schiefer, and Alexander Schatten. Concepts and Models for Typing Events for Event-Based Systems. In DEBS, pages 62 Ű 70. ACM, 2007.

Szabolcs Rozsnyai, Josef Schiefer, and Alexander Schatten. Detecting and Preventing Fraud. In ICDIM07. The Second International Conference on Digital Information Management, 2007.

Szabolcs Rozsnyai, Roland Vecera, Josef Schiefer, and Alexander Schatten. Event Cloud - Searching for correlated business events. In CEC/EEE, pages 409 Ű 420. IEEE Computer Society, 2007 - *BEST PAPER AWARD*.

Josef Schiefer, Szabolcs Rozsnyai, Christian Rauscher, and Gerd Saurer. Event-Driven Rules for Sensing and Responding to Business Situations. In DEBS, pages 198 Ű 205. ACM, 2007.

Roland Vecera, Szabolcs Rozsnyai, and Heinz Roth. Indexing and Search of Correlated Business Events. Ares, pages 1124 Ű 1134, 2007.

9.6 Curriculum Vitae

Personal Information:

Name:	Szabolcs Rozsnyai
Birthday:	16.10.1981
Birthplace:	Cluj-Napoca (Klausenburg)

Education:

since 2006	PhD studies of computer science at University of Technology Vienna
2004-2006	Computer Science - MSc (Information and Knowledge Management) at University of Technology Vienna - *Graduation with Honors*
2004-2007	Computer Science - MSc (Informatikmanagement) at University of Technology Vienna - *Graduation with Honors*
2002-2004	Computer Science - BSc (Software and Information Engineering) at University of Technology Vienna
1996-2001	Secondary College for *data processing and organisation*
1992-1969	Gymnasium with a focus on mathematics and science
1988-1992	Elementary School

Work Experience:

since Dec. 2006	**Senactive IT-Dienstleistungs GmbH** Product development and research in the field of event-based systems and complex event processing.
May 2006 - Dec 2006	**Capgemini Consulting Austria** Engaged as a CRM/DWH consultant working on a CRM/B2B project for an international client in CEE. My responsibilities covered: preparing and performing requirements workshops with international clients in CEE, conducting requirements analyses, coordinating the alignment of involved projects, responsible for test- and training management including coordination of intercultural team members. Further involved in several acquisition efforts.

June 2004 - Dec 2005	**Insilico Software GmbH** Product development of life science and bioinformatics products. Data Integration, Design and Prototyping of a Data Warehouse-LIMS-System for DNA-Microarray experiments. Maintenance and development of the API, the web front-end and the MS Office Plug-in for the MASI (Meta Annotated Sequence Investigation) product. Visualization of clone libraries based on the GeneOntology including references to genes with relevant annotations.
2004	**UNION Versicherungs- AG** Department of IT and Organisation mainly focusing on SW development and maintenance tasks and user support.
2004	**Wolf Office Team** Development of a warehouse/stock management system for a building service engineering company.
2003-2004	**Institute of Software Technology and Interactive Systems** Working on a web-based project management and communication tool that supports (scientific) cooperation, resource management, project monitoring and information distribution.
2002 *3 Months Internship*	**DPW H.R. Software GmbH** Development of a SW prototype that allows a dynamic function and interface generation of an existing software which scales over a Progress database. Development of a SW application that generates organigrams out of given enterprise structures and plots them into the PDF format with a sheet scaling feature.
2001 *21/2 Months Internship*	**Delphi Software GmbH** Revision of Greiner Verpackungen's existing B2B platform.
2000-2001	**Delphi Software GmbH** Webadministrator

1999-2001	**UNION Versicherungs- AG** Department of IT and Organisation mainly focusing on SW development and maintenance tasks and user support. Further development and maintenance of a broker and customer administration tool with a set of reporting and analysis functions.
1999-2001 *1 Month Internship*	**UNION Versicherungs- AG** Department of IT and Organisation. Development of a management system for software and hardware installations.

Bibliography

[1] Daniel Abadi, Donald Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J*, 12(2):120–139, 2003.

[2] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[3] Asaf Adi and Opher Etzion. Amit - the situation manager. *VLDB J*, 13(2):177–203, 2004.

[4] Aguilera, Strom, Sturman, Astley, and Chandra. Matching events in a content-based subscription system. In *PODC: 18th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1999.

[5] ANTLR. Antlr webpage. http://www.antlr.org/, 02 2008.

[6] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, 2001.

[7] Jean Bacon, Ken Moody, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, and Mark D. Spiteri. Generic support for distributed applications. *IEEE Computer*, 33(3):68–76, 2000.

[8] J.M. Bacon and P.R. Pietzuch. Hermes: a distributed event-based middleware architecture. *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, 2002.

[9] J. Bailey, L. Crnogorac, K. Ramamohanarao, and H. Sondergaard. Abstract interpretation of active rules and its use in termination analysis. In *Proceedings of the 6th International Conference on Database Theory (ICDT'97)*, volume 1186 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 1997.

[10] Magdalena Balazinska, Hari Balakrishnan, Jon Salz, and Michael Stonebreaker. The medusa distributed stream-processing system. http://nms.lcs.mit.edu/projects/medusa, 01 2008.

[11] G. Banavar, M. Kaplan, K. Shaw, R. Strom, D. Sturman, and W. Tao. Information flow based event distribution middleware. In *19th International Conference on Distributed Computing Systems (19th ICDCS'99) Workshop on Electronic Commerce and Web-Based Applications*. IEEE, 1999.

[12] Baralis and Widom. An algebraic approach to rule analysis by means of triggering and activation graphs. In *VLDB'94*, 2002.

[13] D. Barbara, S. Mehrotra, and M. Rusinkiewicz. Incas: A computation model for dynamic workflows in autonomous distributed environments. Technical report, 1994.

[14] Tim Bass. Mythbusters: Esp vs. cep. http://thecepblog.com/2007/06/23/debs-2007-mythbusters-esp-v-cep/, 01 2008.

[15] John Bates, Jean Bacon, Ken Moody, and Mark D. Spiteri. Using events for the scalable federation of heterogeneous components. In Paulo Guedes and Jean Bacon, editors, *ACM SIGOPS European Workshop*, pages 58–65. ACM, 1998.

[16] Thomas Bernhardt and Alexandre Vasseur. Esper: Event stream processing and correlation. http://www.onjava.com/lpt/a/6955, 01 2008.

[17] M. Bhandaru, R. Bhatnagar, L. Epshtein, Peng Wu, and Zhongwen Shi. Alarm correlation engine (ace). *Network Operations and Management Symposium, 1998. NOMS 98., IEEE*, 3.

[18] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, 1993.

[19] K. P. Birman, R. Cooper, T. A. Jospeh, K. P. Kane, F. Schmuck, and M. Wood. *ISIS-a distributed programming environment*. Cornell University, Ithaca, NY, June 1990. in User's Guide and Reference Manual.

[20] Robert M. Bruckner, Beate List, Josef Schiefer, and A. Min Tjoa. Modeling temporal consistency in data warehouses. In A. Min Tjoa and Roland Wagner, editors, *DEXA Workshops*, pages 901–905. IEEE Computer Society, 2001.

[21] Christoph Bussler and Stefan Jablonski. Implementing agent coordination for workflow management systems using active database systems. In *RIDE-ADS*, pages 53–59, 1994.

[22] Antonio Carzaniga, Elisabetta Di Nitto, David S. Rosenblum, and Alexander L. Wolf. Issues in supporting event-based architectural styles. In *Third International Software Architecture Workshop*, pages 17–20, Orlando, Florida, November 1998.

[23] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[24] Antonio Carzanigay, David S. Rosenblum, and Alexander L. Wolfy. Challenges for distributed event services: Scalability vs. expressiveness, July 31 1999.

[25] Ceglar, Roddick, and Mooney. From rule visualisation to guided knowledge discovery. *In Proc. Second Australasian Data Mining Workshop, Canberra*, 2003.

[26] Mani Chandy. Event-driven applications: Where they apply and how they are built. www.infospheres.caltech.edu/papers/gartner05.pdf, 11 2007.

[27] Shyh-Kwei Chen, Jun-Jang Jeng, and Henry Chang. Complex event processing using simple rule-based event correlation engines for business performance management. In *CEC/EEE*. IEEE Computer Society, 2006.

[28] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Ying Xing, and Stanley B. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.

[29] C. Cieri. Multiple annotations of reusable data resources: Corpora for topic detection and tracking, 2000.

[30] Esper Codehaus. Esper. http://esper.codehaus.org/, 01 2008.

[31] Esper Codehaus. Esper wiki. http://docs.codehaus.org/display/ESPER/Home, 01 2008.

[32] G. F. Coulouris and J. Dollimore. Addison-Wesley, 2005.

[33] G. Cugola, E. Di Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *Software Engineering, IEEE Transactions on*, 27, 2001.

[34] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *ICSE*, 1998.

[35] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *19 ACM SIGMOD Conf. on the Management of Data, Atlantic City*, 1990.

[36] Wolfgang Emmerich. Software engineering and middleware: a roadmap. In *ICSE - Future of SE Track*, pages 117–129, 2000.

[37] Eugster, Felber, Guerraoui, and Kermarrec. The many faces of publish/subscribe. *CSURV: Computing Surveys*, 35, 2003.

[38] Patrick Eugster. Type-based publish/subscribe: Concepts and experiences. *ACM Trans. Program. Lang. Syst*, 29(1), 2007.

[39] Patrick Th. Eugster and Rachid Guerraoui. Content-based publish/subscribe with structural reflection. In *COOTS*, pages 131–146. USENIX, 2001.

[40] Patrick Th. Eugster, Rachid Guerraoui, and Christian Heide Damm. On objects and events. In *OOPSLA*, pages 254–269, 2001.

[41] Ludger Fiege. *Visibility in Event-Based Systems*. PhD thesis, Technische Universität Darmstadt, 2005.

[42] Ludger Fiege, Gero Mühl, and Alejandro P. Buchmann. An architectural framework for electronic commerce applications. In *GI Jahrestagung (2)*, pages 928–938, 2001.

[43] Martin Fowler. Time point. http://martinfowler.com/eaaDev/TimePoint.html, 03 2006.

[44] Hal Fulton. *Ruby Way*. Addison-Wesley Professional, 2007.

[45] Erich Gamma, Richard Helm, and Ralph Johnson. *Design Patterns*. Addison-Wesley Professional, 1997.

[46] Andreas Geppert and Dimitrios Tombros. Event-based distributed workflow execution with eve. Technical report, Department of Computer Science, University of Zurich, 1996.

[47] Matteo Golfarelli, Stefano Rizzi, and Iuris Cella. Beyond data warehousing: what's next in business intelligence? In Il-Yeol Song and Karen C. Davis, editors, *DOLAP*, pages 1–6. ACM, 2004.

[48] Daniel Gyllstrom, Eugene Wu 0002, Hee-Jin Chae, Yanlei Diao, Patrick Stahlberg, and Gordon Anderson. SASE: Complex event processing over streams. *CoRR*, abs/cs/0612128, 2006.

[49] Richard Hackathorn. Current practices in active data warehousing. *DMReview*, 2002.

[50] Gregor Hohpe. Programming without a call stack - event-driven architectures. www.enterpriseintegrationpatterns.com/docs/EDA.pdf, 11 2007.

[51] IBM. Publish/subscribe over public networks. *Technical Report*, 2001.

[52] C. S. Jensen, J. Clifford, R. Elmasri, S. K. Gadia, P. Hayes, and S. Jajodia [eds]. A glossary of temporal database concepts. *sigmod*, 23(1):52–64, March 1994.

[53] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 1978.

[54] David Luckham. *The Power Of Events*. Addison Wesley, 2005.

[55] David Luckham, Antoine Manens, Sumit Bhansali, Woosang Park, and Susheel Daswani. Modeling and causal event simulations of electronic business processes. 2005.

[56] David Luckham and Roy Schulte. Event processing glossary. http://complexevents.com/?p=195, 01 2008.

[57] Bertram Ludaescher. *Integration of Active and Deductive Database Rules*. PhD thesis, University of Freiburg.

[58] Juha Makkonen, Helena Ahonen-Myka, and Marko Salmenkivi. Applying semantic classes in event detection and tracking. In Rajeev Sangal and S. M. Bendre, editors, *Proceedings of International Conference on Natural Language Process ing (ICON 2002)*, pages 175–183, Mumbai, India, 2002.

[59] Heikki Mannila and Pirjo Moen. Similarity between event types in sequences. In Mukesh K. Mohania and A. Min Tjoa, editors, *DaWaK*, volume 1676 of *Lecture Notes in Computer Science*, pages 271–280. Springer, 1999.

[60] Jean-Louis Maréchaux. Combining service-oriented architecture and event-driven architecture using an enterprise service bus. http://www-128.ibm.com/developerworks/webservices/library/ws-soa-eda-esb/index.html, 11 2007.

[61] René Meier and Vinny Cahill. Taxonomy of distributed event-based programming systems. In *International Workshop on Distributed Event-Based Systems (DEBS'02)*, pages 585–588. IEEE, July 2002. short paper session.

[62] Enterprise Messaging, Bobby Woolf, and Kyle Brown. Patterns of system integration with enterprise messaging, July 24 2002.

[63] G. Mühl, L. Fiege, and A. P. Buchmann. Evaluation of cooperation models for electronic business. In *Information Systems for E-Commerce, Conference of German Society for Computer Science*, pages 81–94, November 2000.

[64] Gregor Mühl, Ludger Fiege, and Alejandro Buchmann. Filter similarities in content-based publish/subscribe systems. *Lecture Notes in Computer Science*, 2299:224–??, 2002.

[65] Pirjo Moen. *Attribute, Event Sequence, and Event Type Similarity Notions for Data Mining.* PhD thesis, University of Helsinki.

[66] Yefim V. Natis. Service-oriented architecture scenario. www.gartner.com/resources/114300/114358/114358.pdf, 11 2007.

[67] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The information bus: an architecture for extensible distributed systems. *SIGOPS Oper. Syst. Rev.*, 27(5):58–68, December 1993.

[68] OMG. Object management group. management of event domains specification. http://www.omg.org/cgi-bin/doc?formal/2001-06-03, 11 2007.

[69] Mark Palmer. Event stream processing - a new physics of software. *DMReview*, 2005.

[70] Claus H. Pedersen. Extending ordinary inheritance schemes to include generalization. In *OOPSLA*, pages 407–417, 1989.

[71] Peter Robert Pietzuch. *Hermes: A Scalable Event-Based Middleware.* PhD thesis, University of Cambridge, 2004.

[72] D. S. Rosenblum and A. L. Wolf. A design framework for Internet-scale event observation and notification. *Lecture Notes in Computer Science*, 1301:344–??, 1997.

[73] Szabolcs Rozsnyai. *Efficient indexing and searching in correlated business events.* PhD thesis, Technische Universität Wien, 2006.

[74] Szabolcs Rozsnyai, Josef Schiefer, and Alexander Schatten. Concepts and models for typing events for event-based systems. In *DEBS*, pages 62–70. ACM, 2007.

[75] Szabolcs Rozsnyai, Josef Schiefer, and Alexander Schatten. Detecting and preventing fraud. In *ICDIM07.* The Second International Conference on Digital Information Management, 2007.

[76] Szabolcs Rozsnyai, Roland Vecera, Josef Schiefer, and Alexander Schatten. Event cloud - searching for correlated business events. In *CEC/EEE*, pages 409–420. IEEE Computer Society, 2007.

[77] Markku Sakkinen. Exheritance - class generalization revived. In *Inheritance Workshop at ECOOP 2002.* ECOOP, 2002.

[78] M. Sayal, F. Casati, U. Dayal, and M. C. Shan. Business process cockpit. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB'02)*, pages 880–883. Morgan Kaufmann, 2002.

[79] Josef Schiefer and Carolyn McGregor. Correlating events for monitoring business processes. In *ICEIS (1)*, pages 320–327, 2004.

[80] Josef Schiefer, Szabolcs Rozsnyai, Christian Rauscher, and Gerd Saurer. Event-driven rules for sensing and responding to business situations. In *DEBS*, pages 198–205. ACM, 2007.

[81] Josef Schiefer and Andreas Seufert. Management and controlling of time-sensitive business processes with sense & respond. In *CIMCA/IAWTIC*, pages 77–82. IEEE Computer Society, 2005.

[82] Marco Seiriö and Mikael Berndtsson. Design and implementation of an ECA rule markup language. In *RuleML*, volume 3791, pages 98–112. Springer, 2005.

[83] David Sternberger. *Evaluierung, Implementierung und Testen von Message Oriented Middleware.* PhD thesis, Technische Universität Wien, 2006.

[84] Robert Strom, Guruduth Banavar, Tushar Ch, Marc Kaplan, Kevan Miller, Bodhi Mukherjee, Daniel Sturman, and Michael Ward. Gryphon: An information flow based approach to message brokering. 2001.

[85] Martin Suntinger, Hannes Obweger, Josef Schiefer, and Groeller. The event tunnel: Interactive visualization of complex event streams for business process pattern analysis. Technical report, Institute of Computer Graphics and Algorithms - Vienna University of Technology, 2007.

[86] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall, 2002.

[87] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous queries over append-only databases. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 321–330, New York, NY, USA, 1992. ACM.

[88] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[89] Roland Vecera. *Efficient indexing, Searching and Analysis of Event Streams.* PhD thesis, Technische Universität Wien, 2007.

[90] Roland Vecera, Szabolcs Rozsnyai, and Heinz Roth. Indexing and search of correlated business events. *Ares*, pages 1124–1134, 2007.

[91] Yiming Yang, Jaime Carbonell, Ralf Brown, Tom Pierce, Brian T. Archibald, and Xin Liu. Learning approaches for detecting and tracking news events, 1999.

[92] Stan Zdonik, Michael Stonebraker, Mitch Cherniack, Ugur C Etintemel, Magdalena Balazinska, and Hari Balakrishnan. The aurora and medusa projects, May 06 2003.

[93] Andreas Zeidler. *Distributed Publish/Subscribe Notification Service for Pervasive Environments*. PhD thesis, Technischen Universität Darmstadt, 2004.

[94] D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In *15th International Conference on Data Engineering (ICDE '99)*, pages 392–399, Washington - Brussels - Tokyo, March 1999. IEEE.

List of Figures