Concepts and Models for Typing Events for Event-Based Systems

Szabolcs Rozsnyai
Secure Business Austria
Favoritenstraße 16
1040 Vienna, Austria
rozsnyai@ securityresearch.at

Josef Schiefer
Institute for Software Technology and Interactive Systems
Favoritenstrasse 9-11/188
1040 Vienna, Austria
js@ifs.tuwien.ac.at

Alexander Schatten
Institute for Software Technology and Interactive Systems
Favoritenstrasse 9-11/188
1040 Vienna, Austria
alexander.schatten@ifs.tuwien.ac.at

ABSTRACT

Event-based systems are increasingly gaining widespread attention for applications that require integration with loosely coupled and distributed systems for time-critical business solutions. In this paper, we show concepts and models for representing, structuring and typing events. We discuss existing event models in the field and introduce the event model of the event-based system SARI for illustrating various typing concepts. The typing concepts cover topics such as type inheritance and exheritance, dynamic type inferencing, attribute types, as well as the extendibility and addressability of events. We show how the typing concepts evolved and depend on the implemented event-based systems which use different approaches for the event processing such as graphical approaches, or approaches, that use Java code, SQL code, or ECA (event-condition-action) rules.

Categories and Subject Descriptors

C.0 Computer Systems Organization: Modeling of computer architecture; System architectures; Systems specification methodology C.2.4 [Computer-Communication Networks]: Distributed Systems; C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems

General Terms

Standardization, Languages, Design

Keywords

Event Model, Complex Event Processing, Event Stream Processing, Service Oriented Architecture

1. INTRODUCTION

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.

The "instant" of an event is relative to the time granularity that is needed or desired. Thus, certain activities that are of short duration relative to the time granularity are represented as a single event. An activity spanning some significant period of time is represented by the interval between two or more events. For example, an order review might have a "begin-review" and "end-review" event-pair. Similarly, a workitem in a workitem list could be represented by the three events "enter-worklist" "start-processing" "end-processing".

For purposes of maintaining information about an action, events also capture attributes about the context when the event occurred. Event attributes are items such as the agents, resources, and data associated with an event, the tangible result of an action (e.g., the result of an approval decision), or any other information that gives character to the specific occurrence of that type of event. Elements of an event context can be used to define a relationship with other events in order to correlate them.

Traditionally, the research on event-based systems has focused 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 system used various ways of representing, filtering and querying events. In many cases, event models have grown from query languages, distributed platforms or architectures for integrating systems.

In this paper, we want to pay attention to existing event models in the field. By defining the scope of an event model, we want to focus on issues for representing and structuring event data.

We introduce the event model of an event-based system called SARI, in order to illustrate and discuss various typing concepts.

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, we want to show the role and importance of an event model 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 autocompletition 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 Mining.** An event model has a significant impact on how event patterns can be discovered in event streams or within historical event traces [17][16]. 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.

2. RELATED WORK

Historically, event models first arose in the context of 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 [13].

Distributed pub/sub architectures such as Hermes [20], Gryphon [4][13], and Siena [6] 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 [19] is a distributed event-based middleware architecture making use of a typed event model and supporting features known from object-oriented programming languages. Further, it has routing algorithms for avoiding global broadcasts and fault tolerance mechanisms. Siena [6] supports restricted event patterns, but it does not define a complete pattern language.

The type-based publish/subscribe was first introduced by Eugster [9][10]. That work attempts to give an event type model that cleanly 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.

Recently, many research projects and industrial solutions work on event stream processing (ESP) and complex event processing (CEP). These approaches address the problem of processing large amounts of events to deliver real-time information, enable closed loop decision making and continuous data integration. In general the approaches allow to monitor, steer and optimize processes in (near) real time.

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

The research for this paper has shown that the underlying event model can provide insights into the capabilities of event processing engines. Despite the discussion 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.

This paper provides an overview of event model concepts and their implementation in various solutions.

Esper [8] 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.

Borealis and Aurora [2] are stream processing engines for SQL-based queries over streaming data with efficient scheduling and QoS delivery mechanisms. Medusa [27] focuses on extending Auroras stream processing engine to distribute the event processing. Borealis extends Aurora's event stream processing engines with dynamic query modification and revision capabilities and makes use of Medusas distributed extensions.

RuleCore [21][25] is an event-driven rule processing engine supporting Event Condition Action (ECA) rules and providing a user interface for rule building and composite event definitions.

Chen et al. [7] 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.

AMIT [3] is an event stream engine whose goal is to provide high-performance situation detection mechanisms. AMIT offers a sophisticated user interface for modelling business situations based on the following four types of entities: events, situations, lifespans and keys. An event is the base entity and can be specified with a set of typed attributes. Further, events can have relationships among each other. 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.

The importance of a solid event model and event typing is also very crucial for event mining applications. Moen describes in [17][16] algorithms which use typed events to determine similarities between event objects in order to discover similar event patterns.

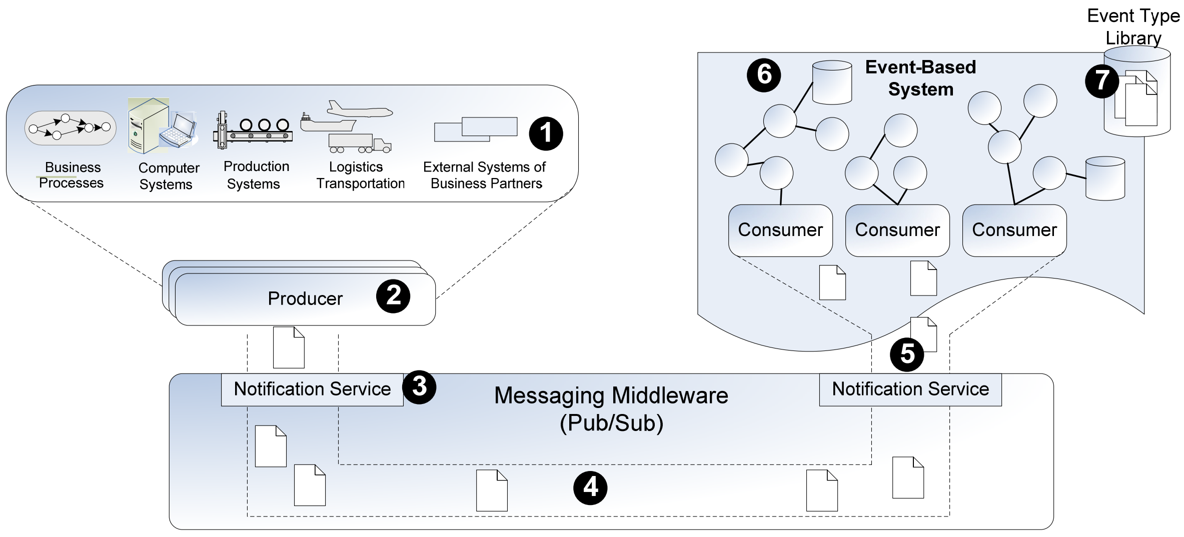


Figure 1. Event-Based Messaging Middleware

3. EVENT-BASED MESSAGING MIDDLEWARE

Client/server and request/reply application architectures, where system components access remote infrastructure and functionality to accomplish their own tasks, have led to the need 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 message-oriented middleware paradigm has become one of the communication pillars of today's enterprise systems. The possibility to exchange messages in distributed heterogeneous environments in an asynchronous persistent way [26] enables a reliable and transparent solution of exchanging data between peers. In contrast to request and reply patterns, this communication paradigm provides an abstraction of the underlying network complexity, 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.

Messaging solutions and, in particular, event-based systems are characterized by having two interacting components — the consumer and the producer. Figure 1 illustrates the architecture of a distributed messaging solution that connects producers and consumers through a message-oriented pub/sub middleware. The *producer* (2) takes relevant observations into account and decides if they should be published. The notification service, respectively the message-oriented middleware, 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 in events by subscribing to their notifications. The subscribed peers, the *consumers* (5), react on delivered event notifications. *Event-based systems* (6) provide adapters which are consumers of event notification 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* (3) is the core enabler for loosely coupled systems since they provide a transparent abstraction layer for programming languages that hide the communication details beneath. The notification services task is to deliver every published notification to all consumers that indicated an interest (e.g. created a subscription). A *channel* (4) represents a bundle of notifications that a consumer can subscribe to or a producer could select to distribute its notifications. *Event sources* (1) can be a wide range of independent systems. For instance, it can be a production-, a logistics- or some external business partner system.

Events are defined as observable actions or relevant state changes in IT systems [11][14][15]. A relevant state change is always a subjective occurrence and so are events. The representation of information about the occurred activity is usually 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. Another aspect of the events' relevance to interested parties is their granularity level. A problem can also be that they are too fine grained, and thus don't deliver enough information. This results in detailed information that is not relevant for the consumers.

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 can not find the requested document, or that the server is not able to process the request. 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.

The *producer* publishes events following a specific syntax and semantics. The *consumer* subscribes to certain notifications of events that are created by observing interesting occurrences. 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.

4. EVENT MODEL CONCEPTS

Decoupling producers and consumers leads to a number of advantages previously discussed. 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 un-typed events in the event model requires a strategy to deal with potentially missing event attributes and ambiguous event types.

We define the scope of an event model 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.

The next chapters will introduce various event model concepts and illustrate them with examples from the SARI system [24]. Examples for defining relationships between events in SARI can be found in [23] and is therefore out of the scope for this paper.

4.1 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 2 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.

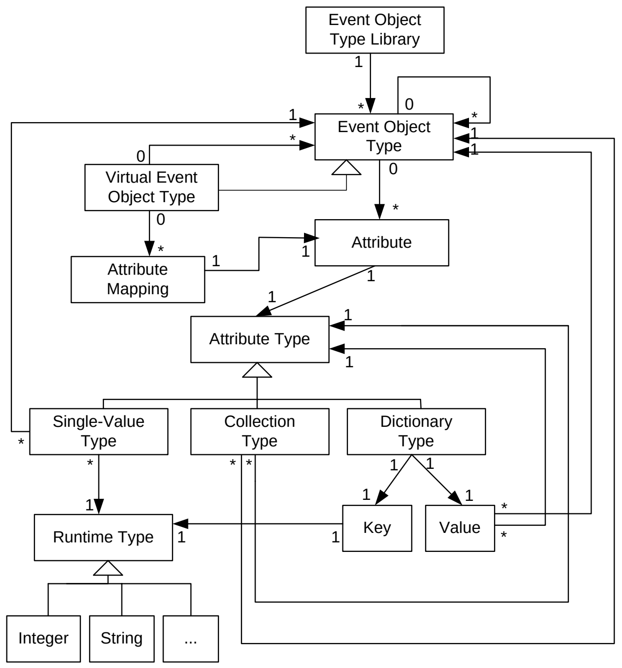


Figure 2. Event Object Type Library Model

At the lowest level, an attribute value has to correspond to an runtime type (Integer, String, …). A more advanced concept that can be realized within SARI is exheritance where virtual event objects can be created with attributes that are mapped to attributes of existing event object types. Exheritance 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).

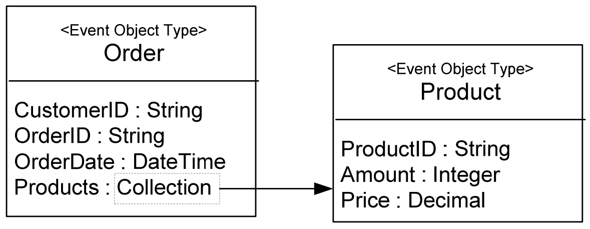


Figure 3. Event Object Type Example

Figure 3 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.

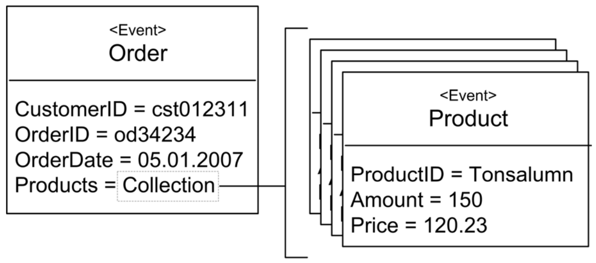


Figure 4. Event Object at Runtime

Figure 4 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.

4.2 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 supported event attribute types through an example presented in Figure 5.

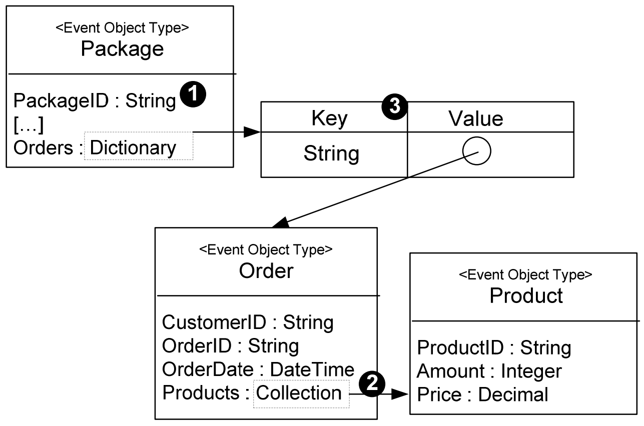


Figure 5. Attribute Model Example

Single-Value Types

Single-value types represent attributes which have a runtime 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 (Point 1) shows the attribute PackageID which is a single-value type of type string.

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 (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 (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 possible 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.

4.3 Inheritance

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.

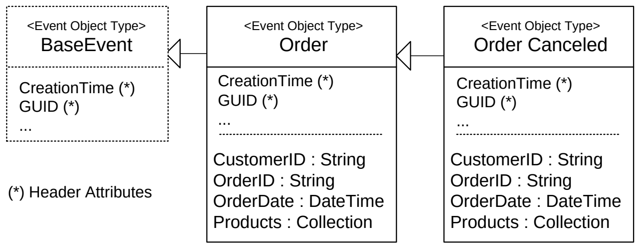


Figure 6. Inheritance Example

Figure 6 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.

4.4 Exheritance

Another supported concept is exheritance which the possibility to create generalizations from event object types (see also [22] and [18]). 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 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. Figure 7 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 Activity Started event object type and ShipmentDelivered form a new ActivityCompleted event object type.

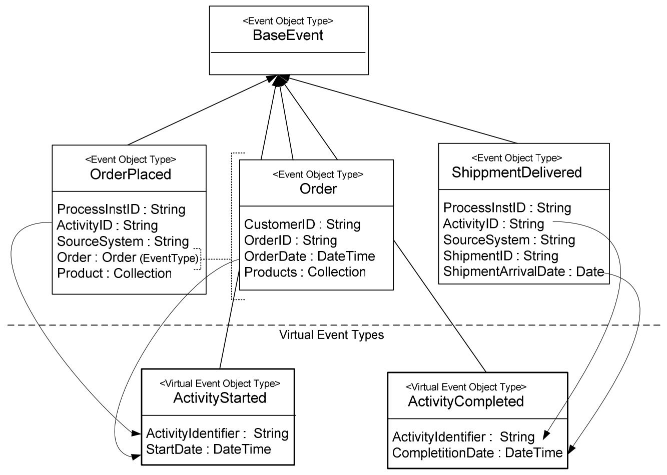


Figure 7. Exheritance Example

4.5 Duck Typing

The concept of *duck typing* goes back to the support of dynamic typing in programming language concepts. Duck typing (*If it walks like a duck and quacks like a duck, it must be a duck* [12]) allows to implement interfaces dynamically at runtime and is part of programming languages like Ruby and Python.

The idea behind duck typing for events is to allow event services to process un-typed event objects. By un-typed events we mean 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.

Figure 8 shows an example of duck typing in SARI. SARI uses event processing maps (see Figure 8) for processing event objects with *event services* (represented as rounded rectangles in Figure 8). *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 8 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 8 shows the input ports for event services whose ports expect a 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.

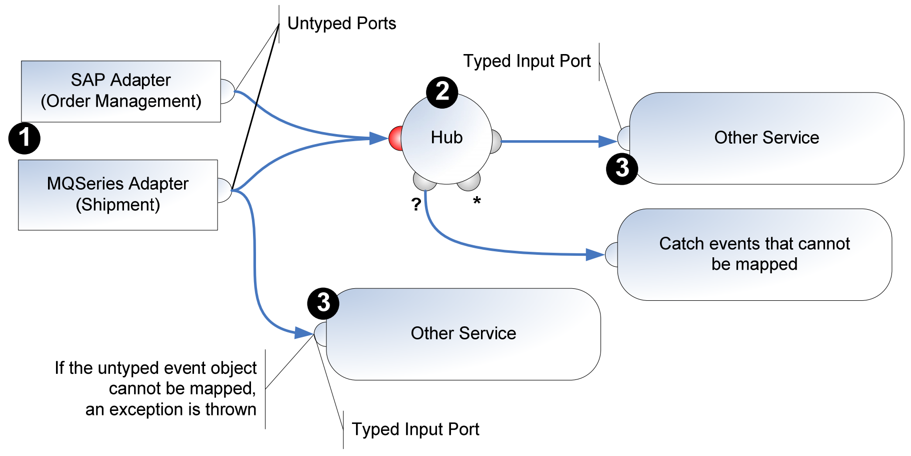
5. EXTENSIBILITY

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.

6. NAMESPACES AND ADDRESSING

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.



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

Figure 8. Duck Typing in an Event Processing Map

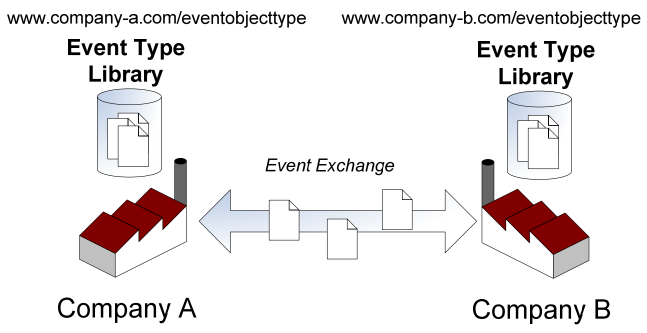


Figure 9. Sharing Event Object Types

Figure 9 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 our 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.

7. 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:

- We developed event access (EA) expressions which have been designed for business users in order to flexibly access event data and to perform calculations. For instance, SARI uses EA expressions for defining rules and for mapping event data to database tables, as well as for defining metrics and filters for analysis purposes.
- 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 on 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, we list typical examples 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, we show some typical examples for application scenarios 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") {…} }

8. COMPARISON OF EVENT MODELS

In the following section, respectively in Table 1, we compare event type models of several existing event processing systems. We analyzed the event model of 6 event-based systems regarding the introduced event typing concepts in this paper. We selected systems that consider themselves as complex event processing and event stream processing systems, with the intension 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.

	Event Type Model	Attribute Model	Inheritance/Exheritance	Advanced Typing Concepts	Addressing	Event Access	Event Representation
SARI	• Repository for event type definitions • Globally addressable event types • Event object type definition in XML • Validation through XSD	• Single-value types • Collection types • Dictionary types • Support of unknown attributes	• Inheritance supported • Exheritance supported	Duck Typing	• URI	• EAExpressions • XPath • Programmatically	• Java Objects • .Net Objects • XML
Esper	• Event object type definition in XML • Supports structured events • Validation through XSD	• Attribute types are determined by POJOs (Plain old Java objects) following the JavaBean conventions • Dictionary and collection types are supported	• Inheritance supported through OO concepts in Java	Event aliasing	• Namespace implicitly available through Java	• SQLlike syntax • XPath • Programmatically	• Java Objects • XML
Hermes	• Event object type definition in XML • Validation through XSD	• XML representation that can be bound to Java classes	• Inheritance supported	-	• XML namespace for avoiding naming conflicts	• Programmatically • XPath	• Java Objects
Borealis	• Schema defines valid tuples • Validation through XSD	• Single-value types supporting number, date and string values	-	-	-	• SQLlike syntax	• C++ structs
RuleCore	• Event object type definition in XML • Validation through XSD	• Single-value types supporting number, date and string values	-	-	• Can be enforces implicitly	• XPath	• Python classes
AMIT	• Event type definition in XML • Validation through XSD	• Single-value types supporting number, date and string values	• Inheritance supported	-	• Event types have an identifier	• SQLlike syntax	• Java Objects • XML

Table 1. Comparison of Existing Event-Based Processing Solutions

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.

9. CONCLUSION

As event-based systems are increasingly gaining widespread attention we addressed the problem that those systems lack in the support of tools that allow users to easily reconfigure a system or to refactor services and components. Event models have a major impact on the flexibility and usability. Therefore, we introduced concepts and approaches for representing, structuring and typing event data and introduced event models of existing event-based solutions. We discussed concepts of organizing event models, we introduced basic typing concepts for structuring event data, we also introduced more advanced typing concepts such as inheritance, exheritance and dynamic type inferencing. We illustrated the typing concepts with the event-based system SARI and compared them with existing event stream processing systems.

This paper presents a long-term research effort aiming at consolidating and creating a unified event model for event-based systems. The long term goal is the development of a rich event-model which can be supported by a wide range of event-based systems.

10. REFERENCES

[1] Abadi, D. J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N. and Zdonik, S. Aurora: A new model and architecture for data stream management. VLDB Journal 12, pp. 120–139, August 2003.

[2] Abadi, D.J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The Design of the Borealis Stream Processing Engine. In: Proc. of the Conf. on Innovative Data Systems Research, Asilomar, CA, USA (2005) 277–289.

[3] Adi, A. and Etzion, O. (2004). AMIT - the situation manager. The VLDB Journal, 13(2):177.203.

[4] Banavar, G., Kaplan, M., Shaw, K., Strom, R., Sturman, D., and Tao, W. Information flow based event distribution middleware. In Proceeding of ICDCS'99 Middleware Workshop, 1999.

[5] Carney, D., Cetintemel, U., and et al. Monitoring streams - a new class of data management applications. In Proc. of VLDB, 2002.

[6] Carzaniga, A., Rosenblum, D.S.,Wolf, A.L.: Design and Evaluation of a Wide-Area Event Notification Service. ACM Trans. on Comp. Sys. 19 (2001) 332-383.

[7] Chen, S.K., Jeng, J.J, Chang, H., Complex Event Processing using Simple Rule-based Event Correlation Engines for Business Performance Management. CEC/EEE 2006.

[8] Esper, http://esper.sourceforge.net, 2007-03-10.

[9] Eugster, P. T. and Guerraoui, R. Content-Based Publish/ Subscribe with Strucutural Reflection. In Proc. of the 6th USENIX Conf. on Object-Oriented Technologies and Systems (COOTS01), Jan. 2001.

[10] Eugster, P. T., Guerraoui, R., Sventek, J.. Type-Based Publish/Subscribe. Technical report, EPFL, Lausanne, Switzerland, June 2000.

[11] Fiege, L. Visibility in Event-Based Systems. PhD thesis, Technische Universität Darmstadt, 2005.

[12] Funton, H. Ruby Way. Addison-Wesley Professional, 2007.

[13] IBM T J Watson Research Center: Gryphon: Publish/Subscribe Over Public Networks. Whitepaper (2002).

[14] Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21, 1978.

[15] Luckham, D. The Power Of Events. Addison Wesley, 2005.

[16] Mannila, H., Moen, P. Similarity between event types in sequences, Proc. First Intl. Conf. on Data Warehousing and Knowledge Discovery (DaWaK'99), Florence, Italy, 1999, pp. 271–28.

[17] Moen, P., Attribute, Event Sequence, and Event Type Similarity Notions for Data Mining, Ph.D. thesis, Department of Computer Science, University of Helsinki, Finland, 2000.

[18] Pedersen, C.H. Extending ordinary inheritance schemes to include generalization. In Conference proceedings on Object-oriented programming systems, languages and applications, pages 407-417. ACM Press, 1989.

[19] Pietzuch, P. R. and Bacon, J., Hermes: A Distributed Event-Based Middleware Architecture, Proceedings of the 22nd International Conference on Distributed Computing Systems, p.611-618, July 02-05, 2002.

[20] Pietzuch, P.R., Bacon, J.M.: Hermes: A Distributed Event-Based Middleware Architecture. In: Proc. of the 1st Int. Workshop on Distributed Event-Based Systems (DEBS'02), Vienna, Austria (2002) 611-618.

[21] RuleCore, http://www.rulecore.com/, 2007-03-10.

[22] Sakkinen, M. Exheritance – Class Generalization Revived. In Proceedings of the Inheritance Workshop at ECOOP 2002, Malaga, Spain, June 2002.

[23] Schiefer, J. and McGregor, C., Correlating events for monitoring business events, ICEIS 2004, 2004.

[24] Schiefer, J., Seufert, A., Management and Controlling of Time-Sensitive Business Processes with Sense & Respond, International Conference on Computational Intelligence for Modelling Control and Automation (CIMCA), Vienna, 2005.

[25] Seirio, M., Berndtsson, M. Design and Implementation of an ECA Rule Markup Language. *RuleML*, Springer Verlag, pp. 98–112, 2005.

[26] Tanenbaum A., van Stehen, M. Distributed Systems. Principles and Paradigms. Prentice Hall, 2003.

[27] Zdonik, S., Stonebraker, M., Cherniack, M., Cetintemel, U., Balazinska, M., Balakrishnan, H.. The Aurora and Medusa Projects. IEEE Data Engineering Bulletin, 26, March 2003.