Integration of Event Processing with Service-oriented Architectures and Business Processes

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte Dissertation zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
von M.Sc. Stefan Appel aus Friedberg (Hessen)
26. März 2014 — Darmstadt — D 17

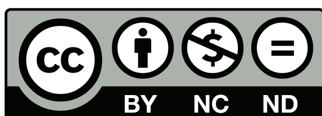
TECHNISCHE
UNIVERSITÄT
DARMSTADT

DVS

1. Gutachter:	Prof. Alejandro Buchmann, Ph.D.	Tag der Einreichung:	26. März 2014
2. Gutachter:	Jun.-Prof. Dr.-Ing. Dimka Karastoyanova	Tag der Prüfung:	20. Mai 2014
3. Gutachter:	Prof. Patrick Eugster, Ph.D.		

Bitte zitieren Sie dieses Dokument als:
URN: urn:nbn:de:tuda-tuprints-39785
URL: http://tuprints.ulb.tu-darmstadt.de/3978

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de



Die Veröffentlichung steht unter folgender Creative Commons Lizenz:
Namensnennung – Keine kommerzielle Nutzung – Keine Bearbeitung 3.0 Deutschland
http://creativecommons.org/licenses/by-nc-nd/3.0/de/

Curriculum Vitae

Stefan Appel

Personal Information

Date of Birth	November 1, 1982
Place of Birth	Friedberg (Hessen), Germany

Education

2009–2014	PhD Student **Databases and Distributed Systems Group, TU Darmstadt**
2006-2009	M.Sc. Computer Science **TU Darmstadt**
2007-2008	M.Sc. Computer Science **University of Colorado at Boulder**
2003-2006	B.Sc. Computer Science **TU Darmstadt**
2002	Abitur **Augustinerschule Friedberg (Hessen)**

Positions

2009–2013	Research and teaching assistant **Databases and Distributed Systems Group, TU Darmstadt**
2012	Visiting Scholar **Purdue Universtiy**

Zusammenfassung

Datenquellen wie das Internet der Dinge oder Cyber-physical Systems stellen enorme Mengen an Echtzeitinformationen in Form von Ereignisströmen zur Verfügung. Solche Ereignisströme ermöglichen die Entwicklung reaktiver Software Komponenten die als Bausteine einer neuen Generation von Systemen dienen. Unternehmen können beispielsweise neue Dienstleistungsangebote bereitstellen, die auf der Nutzung von Ereignisströmen basieren. Ebenso können bestehende Geschäftsprozesse durch die Integration von Echtzeitdaten in Form von Ereignisströmen optimiert werden. Die Integration von Komponenten zur Ereignisstromverarbeitung in bestehende Anwendungslandschaften stellt jedoch eine große Herausforderung dar. Während traditionelle Systemkomponenten wie Services einem Abfrage-orientierten Interaktionsstil folgen (Request/Reply), sind Interaktionen in Ereignis-basierten Systemen datengetrieben: Ereignisse treffen kontinuierlich ein – daher die Bezeichnung Ereignisstrom – und führen zum impliziten Aufruf von Anwendungslogik.

In reaktiven Systemen von morgen spielt die Integration beider Paradigmen eine tragende Rolle. Voraussetzung hierfür sind Abstraktionsmechanismen die Ereignisverarbeitung im Kontext komplexer Systemlandschaften kapseln. In dieser Arbeit wird solch eine Abstraktion eingeführt: Ereignisstromverarbeitungseinheiten (Event Stream Processing Units, SPUs) sind ein Container Modell das die Kapselung von Anwendungslogik zur Ereignisverarbeitung sowohl auf der technischen als auch auf der Geschäftsprozessebene ermöglicht. Auf der technischen Ebene stellen SPUs eine serviceähnliche Abstraktion dar, die eine Entwicklung skalierbarer reaktiver Anwendungen vereinfacht. Auf der Geschäftsprozessebene erlauben SPUs eine explizite Darstellung von Ereignisverarbeitung und des damit einhergehenden reaktiven Verhaltens in Prozessen. SPU unterliegen einem Lebenszyklus und werden implizit - bei der Ankunft der entsprechenden Ereignisse - oder explizit - auf Anfrage hin - instanziiert. In Prozessmodellen ermöglicht die Kapselung von Ereignisverarbeitung in SPUs eine direkte Abbildung von Prozessschritten auf IT Komponenten.

Ziel dieser Arbeit ist die umfassende Darstellung des SPU Container Modells. Zunächst wird die Struktur von SPU Containern dargestellt und die Ausführungssemantik definiert. Da SPUs auf ein Publish/Subscribe System zum Transport von Daten zurückgreifen, spielen Quality of Service Aspekte dieser Transportschicht eine zentrale Rolle und werden diskutiert. Des Weiteren sind in Ereignis-basierten Systemen Ereignisproduzenten und -konsumenten logisch entkoppelt, d.h. Produzenten wissen nichts über potentielle Konsumenten und umgekehrt. Dies hat maßgeblichen Einfluss auf den Softwareentwicklungsprozess und erfordert ein adaptiertes Vorgehensmodell. Im Rahmen dieser Arbeit wird daher ein Verfahren zur Anforderungsanalyse vorgestellt das die Charakteristika Ereignis-basierter Systeme berücksichtigt.

SPUs können Ereignisverarbeitung auf der Abstraktionsebene geschäftsrelevanter Vorgänge kapseln und ermöglichen somit eine nahtlose Integration in Geschäftsprozesse. Zur Darstellung von SPUs in Prozessmodellen führt diese Arbeit Erweiterungen der Prozessmodellierungssprachen BPMN und Ereignis-gesteuerte Prozessketten (EPK) ein. Weiterhin wird ein Vorgehen beschrieben wie Prozessmodelle, die SPUs enthalten, zur Ausführung gebracht werden können.

Da das SPU Container Modell sprachunabhängig ist, werden im Rahmen dieser Arbeit Eventlets – eine Implementierung des SPU Container Modells basierend auf Java Enterprise Technologie – vorgestellt. Eventlets werden von einer verteilten Middleware ausgeführt. Die Evaluation zeigt, dass sie den Entwicklungsaufwand skalierbarer Anwendungen zur Ereignisverarbeitung erheblich reduzieren. Da SPUs eine zusätzliche Abstraktionsebene darstellen, wird die Performanz analysiert: Durchsatzmessungen bei der Verarbeitung von Ereignisströmen zeigen, dass Eventlets gegenüber traditionellen Ereignisverarbeitungsansätzen konkurrenzfähig sind.

Abhängig vom Anwendungsszenario kann die Verarbeitung sensibler Daten durch SPUs notwendig sein, z.B. die Verarbeitung von Patientendaten Gesundheitswesen. Daher wird im Rahmen dieser Arbeit die Anwendung kryptografischer Mechanismen zum Schutz der Privatsphäre bei der Zustellung von Ereignissen an SPUs skizziert. Weiterhin werden die damit einhergehenden Performanzeinbußen quantifiziert.

Abstract

Data sources like the Internet of Things or Cyber-physical Systems provide enormous amounts of real-time information in form of streams of events. The use of such event streams enables reactive software components as building blocks in a new generation of systems. Businesses, for example, can benefit from the integration of event streams; new services can be provided to customers, or existing business processes can be improved. The development of reactive systems and the integration with existing application landscapes, however, is challenging. While traditional system components follow a pull-based request/reply interaction style, event-based systems follow a push-based interaction scheme; events arrive continuously and application logic is triggered implicitly.

To benefit from push-based and pull-based interactions together, an intuitive software abstraction is necessary to integrate push-based application logic with existing systems. In this work we introduce such an abstraction: we present Event Stream Processing Units (SPUs) – a container model for the encapsulation of event-processing application logic at the technical layer as well as at the business process layer. At the technical layer SPUs provide a service-like abstraction and simplify the development of scalable reactive applications. At the business process layer SPUs make event processing explicitly representable. SPUs have a managed lifecycle and are instantiated implicitly – upon arrival of appropriate events – or explicitly upon request. At the business process layer SPUs encapsulate application logic for event stream processing and enable a seamless transition between process models, executable process representations, and components at the IT layer.

Throughout this work, we focus on different aspects of the SPU container model: we first introduce the SPU container model and its execution semantics. Since SPUs rely on a publish/subscribe system for event dissemination, we discuss quality of service requirements in the context of event processing. SPUs rely on input in form of events; in event-based systems, however, event production is logically decoupled, i.e., event producers are not aware of the event consumers. This influences the system development process and requires an appropriate methodology. Fur this purpose we present a requirements engineering approach that takes the specifics of event-based applications into account.

The integration of events with business processes leads to new business opportunities. SPUs can encapsulate event processing at the abstraction level of business functions and enable a seamless integration with business processes. For this integration, we introduce extensions to the business process modeling notations Business Process Model and Notation (BPMN) and Event-driven Process Chains (EPCs) to model SPUs. We also present a model-to-execute workflow for SPU-containing process models and implementation with business process modeling software.

The SPU container model itself is language-agnostic; thus, we present Eventlets as SPU implementation based on Java Enterprise technology. Eventlets are executed inside a distributed middleware and follow a lifecycle. They reduce the development effort of scalable event processing applications as we show in our evaluation. Since the SPU container model introduces an additional layer of

abstraction we analyze the overhead in terms of performance and show that Eventlets can compete with traditional event processing approaches in terms of performance.

SPUs can be used to process sensitive data, e.g., in health care environments. Thus, privacy protection is an important requirement for certain use cases and we sketch the application of a privacy-preserving event dissemination scheme to protect event consumers and producers from curious brokers. We also quantify the resulting overhead introduced by a privacy-preserving brokering scheme in an evaluation.

Acknowledgements

Whilst I was working on my master's thesis as part of the Databases and Distributed Systems (DVS) group, the opportunity to pursue a PhD was offered to me. I am glad to have taken this opportunity, as I have had a great time over the past five years, both working within the DVS group and being in Darmstadt. I am very grateful for all of the support that I have received during my PhD, which has made the completion of this thesis possible.

I would like to express gratitude to my advisor Professor Alejandro Buchmann; he always supported the pursuit of my own ideas and put his faith into my work. I thank Junior Professor Dimka Karastoyanova and Professor Patrick Eugster for reviewing my work, as well as Professors Reiner Hähnle, Mira Mezini, and Karsten Weihe for being members of my dissertation defense committee.

I owe thanks to all of my colleagues from the DVS group, with special thanks going to Kai Sachs who motivated me to join DVS as a PhD student and with whom I have worked on many interesting performance evaluation projects.

Thanks to my event-based colleagues, Sebastian Frischbier, Tobias Freudenreich, Pablo Guerrero, and Ilia Petrov, whom I have collaborated with for numerous research and teaching activities; to Astrid Endres, Robert Gottstein, and Khalid Nawaz for being excellent office mates; to the peer-to-peer guys, Alexander Frömmgen, Max Lehn, Christof Leng, and Robert Rehner, who I often visited for procrastination purposes; to Daniel Bausch for never losing countenance when I was complaining about the DVS infrastructure; to Christian Seeger for many interesting discussions about patient-monitoring use cases; and to Maria Braun, Gabriele Ploch, and Maria Tiedemann for their support in all administrative challenges.

During my PhD studies, I closely collaborated with Software AG. This started with the ADiWa project and continued within the scope of the Software Campus initiative; the information exchange with Software AG was always fruitful. Special thanks to Walter Waterfeld for his continued support and valuable feedback; Walter succeeded in defeating many software license issues and made access to Software AG products and services possible.

I would like to express my appreciation to Professor Elisa Bertino and to Nabeel Mohammed for providing me with the opportunity to spend time at Purdue University as a visiting scholar, collaborating on the privacy-preserving publish/subscribe scheme.

I also thank my undergraduate and master's students for the work they put into their theses. Special recognition goes to Pascal Kleber for mastering Event-driven Process Chains.

Thanks to all of the friends that I have met throughout my time in Darmstadt; you made me feel at home!

Finally, thanks to my family. Without their endless support and encouragement, this work would not have been possible.

List of Abbreviations

AMQP Advanced Message Queuing Protocol

BPEL Business Process Execution Language

BPM Business Process Management

BPMN Business Process Model and Notation

CEP Complex Event Processing

CIM Computation Independent Model

CPS Cyber-physical System

DBMS Database Management System

DDS Data Distribution Service

DEBS Distributed Event-based System

DSMS Data Stream Management System

EBC Event-based Component

EBS Event-based System

ED-SOA Event-driven Service-oriented Architecture

EDA Event-driven Architecture

EJB Enterprise Java Bean

EPC Event-driven Process Chain

ERP Enterprise Resource Planning

ESB Enterprise Service Bus

ESPS Event Stream Processing Service

ESPT Event Stream Processing Task

ESS Event Stream Specification

IoT Internet of Things

JMS Java Message Service

MDB Message-driven Bean

MOM Message-oriented Middleware

OASIS Organization for the Advancement of Structured Information Standards

OMG Object Management Group

OOP Object-oriented Programming

QoI Quality of Information

QoS Quality of Service

SLA Service-level Agreement

SOA Service-oriented Architecture

SPU Event Stream Processing Unit

STOMP Simple Text Oriented Messaging Protocol

TTP Trusted Third Party

XPDL XML Process Definition Language

Contents

1 Introduction **1**
1.1 Problem Statement 1
1.2 Motivating Example: Shipment Monitoring 2
1.3 Contributions 3
1.4 Related Activities and Publications 5
1.5 Structure and Terminology 7

2 Background **9**
2.1 Event-Based Systems 9
2.1.1 Event Dissemination with Publish/Subscribe 9
2.1.2 Publish/Subscribe Middleware 12
2.1.3 Event Processing 15
2.1.4 Event-based Interaction 17
2.2 Business Process Modeling and Execution 19
2.2.1 Business Process Modeling with BPMN 19
2.2.2 Business Process Execution 20

3 Event Stream Processing Units **23**
3.1 Event Bus 23
3.2 SPUs as Containers for Generic Event-driven Tasks 25
3.2.1 SPUs as Abstraction to Event Stream Processing 27
3.2.2 SPU Instantiation and Completion Strategies 27
3.2.3 SPU Definition and Structure 28
3.2.4 SPU Execution Semantics 32
3.3 Quality of Service Considerations 34
3.4 Network of SPUs 38
3.5 Instantiation based upon Complex Conditions 40
3.6 Summary 42

4 Requirements Engineering in EBSs **43**
4.1 Event Detection and Publication 44
4.2 Requirements Engineering Case Study 46
4.2.1 Order-to-Delivery Process for Temperature-Sensitive Goods 46
4.2.2 Fleet Management 48
4.2.3 Order-to-Delivery Process with Monitoring of Environmental Conditions 49
4.2.4 Cost Assessment 50
4.3 Implementation Effort and Reuse Potential of Event Publishers 51
4.4 Requirements Engineering Process 52
4.5 Summary 54

5 SPU Integration with Business Processes 57
5.1 Event Stream Integration Requirements 58
5.1.1 Business Process Modeling Layer 58
5.1.2 Workflow Execution Layer 59
5.1.3 IT Infrastructure Layer 59
5.2 Event Stream Processing Units in Business Processes 60
5.2.1 Modeling Layer 61
5.2.2 Workflow Execution Layer 72
5.2.3 IT Infrastructure Layer 77
5.3 Implementation 77
5.4 Summary 79

6 Eventlets: An Implementation of SPUs 81
6.1 Eventlet Middleware Architecture 81
6.1.1 Eventlet Middleware Components 82
6.1.2 Runtime Behavior 83
6.2 Eventlet Middleware Implementation 88
6.2.1 Eventlet Middleware Interfaces 94
6.3 Privacy Concept 94
6.4 Transformation Concept 99
6.5 Summary 102

7 Evaluation 105
7.1 Scalability Benefits with Eventlets 105
7.1.1 Setup and Scenarios 105
7.1.2 Throughput Measurement Results 110
7.1.3 Simplified Software Development with Eventlets 112
7.2 Privacy Preserving Pub/Sub Evaluation 113
7.2.1 Test Environment and Setup 113
7.2.2 Evaluation Results 114
7.3 Summary 117

8 Related Work 119
8.1 Work Related to the SPU Container Model 119
8.2 Related Work to Requirements Engineering in Event-based Systems 120
8.3 Related Work to Event Processing in Business Processes 121
8.4 Related Work to Eventlets 123
8.5 Summary 124

9 Conclusion 127

10 Future Research 131
10.1 Conceptual Layer 131
10.2 Technical Layer 132

1 Introduction

Driven by the advances in information and communication technology, the amount of information available in IT systems steadily increases. Data can be captured and accessed anytime and anywhere. For example, the omnipresence of mobile devices allows the realization of Cyber-physical Systems (CPSs) or the Internet of Things (IoT). Each device can be a data source; the amount of data providers increases. Further, data is collected in real-time. The challenge is the integration of this data with existing IT systems: the data has to be transported from producers to interested consumers. Consumers process the data, e.g., to improve business processes or to provide new services to customers by integrating real-time information from a multitude of sources.

A common paradigm for the representation of information from sources like the IoT or CPSs, are events and streams of events. The notion of a *stream* illustrates, that new events occur continuously over time. In such Event-based Systems (EBSs), event producers do not necessarily know the event consumers, or whether the events will be consumed at all. This independence is intrinsic to the event-based approach [35]. The decoupling of event producers and consumers as well as the arrival of an indefinite number of events over time requires an appropriate event dissemination mechanism. Commonly, publish/subscribe (pub/sub) systems are used; they allow asynchronous communication between fully decoupled participants. Event consumers specify their interest in events in form of *subscriptions*. Optionally, event producers specify the type of events they may publish in *advertisements*.

Pub/sub systems are often used as event dissemination mechanism in event stream processing applications, e.g., in Event-driven Architectures (EDAs) [47,109,117]. These applications receive streams of events and implement reactive behavior, i.e., application logic is invoked implicitly upon arrival of events. From a software engineering perspective, the development of event stream processing applications still happens at a low abstraction level. The goal of this thesis is to raise this level of abstraction by providing building blocks to encapsulate event stream processing application logic. The demand for such a higher-level perspective on event processing was also identified as challenge in the 2010 Dagstuhl seminar on event processing [48].

1.1 Problem Statement

Over the last years, event stream processing has become an established technology. Many vendors provide complex event processing (CEP) engines and event dissemination is realized with message-oriented middleware. However, the integration of event stream processing with large applications is still at an early stage. There are no abstractions to event stream processing that encapsulate event stream processing logic in manageable units that can be integrated with existing (enterprise) applications.

The history of traditional databases systems is one example that illustrates the demand for such encapsulation and abstraction mechanisms. In database management systems (DBMS), data is persistently stored in tables; queries are used to pull data from tables for processing. With more complex

IT architectures and new demands in terms of scalability, usability, reusability, and manageability, abstraction layers were introduced to encapsulate DBMS queries in containers. Developers use these containers as building blocks. For the use of a container, it is important what it does; how it is implemented should be the concern of the container developers only. This separation of concerns is essential to the development of complex applications. Two examples are the persisting of objects and services of a service oriented architecture.

To make objects persistent, the data of object instances is stored in a database. The details are often transparent to developers; they use objects as abstraction mechanism for entities. The same holds for services: rather then writing a database query, developers rely on services that represent business functions. Business functions, e.g., process credit card payment, describe functionality in a declarative way, which facilitates software development. The general goal of abstraction and encapsulation techniques and layers is a separation of concerns. Technical details, e.g., database queries or objects, are implemented by dedicated developers. Other developers can rely on this functionality and use the encapsulated application logic, like objects or services, as building blocks in applications. Entities and entity interactions are in the foreground rather than low-level database queries or object implementations.

Object-oriented programming and layers that encapsulate access to databases became essential. However, in event stream processing, such abstraction layers are missing. Conceptually, event stream processing should be encapsulated in containers that exhibit object- and service-like properties. In this work, we introduce *Event Stream Processing Units (SPUs)* as such an encapsulation and abstraction mechanism as shown in Figure 1.1. SPUs add an abstraction layer to event stream processing functionality that makes it manageable and intuitively usable in complex applications. SPUs are containers with a managed lifecycle that do not aim to replace existing event processing approaches. They allow a clear separation of concerns and hide details. Just like objects and services have proven to be suitable building blocks that encapsulate logic, we pursue the same goal with SPUs in event-based applications.

1.2 Motivating Example: Shipment Monitoring

An example application scenario for SPUs is the monitoring of shipments. When shipments contain temperature-sensitive goods, a constant monitoring of the transport process is desirable to detect problems as soon as possible. Avoiding the decay of goods is advantageous from the logistics provider perspective as well as from the customer perspective.

For such a temperature monitoring application, the trucks are equipped with sensors that continuously report the temperature: streams of temperature events are created [182]. With SPUs it is possible to develop a shipment monitoring application in an intuitive way: an SPU encapsulates the temperature monitoring logic for a shipment. At runtime an SPU instance is created for each shipment. This SPU instance receives the temperature monitoring events associated with a particular shipment and performs the shipment-monitoring task. This can involve looking up thresholds in a database as well as calculating the average temperature throughout the transport. Such a monitoring SPU can also be integrated in business processes so that the process flow changes in case of temperature threshold violations. Further, the monitoring SPU has a managed lifecycle and is stopped as soon as the shipment is delivered.

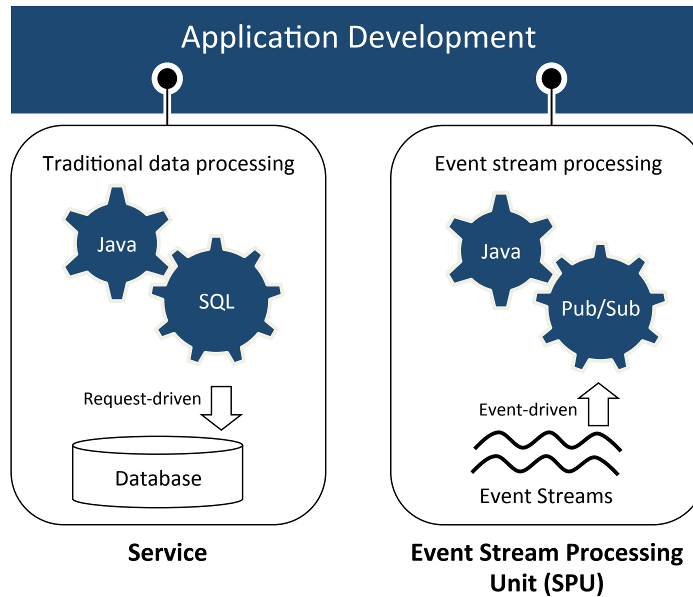


Figure 1.1: Event Stream Processing Units (SPUs) provide a service-like encapsulation of event processing.

1.3 Contributions

In this thesis we introduce Event Stream Processing Units (SPUs) for the encapsulation of event stream processing. We see SPUs as a software engineering concept, more specific, as a container model. We developed SPUs in the context of enterprise applications to integrate and manage event stream processing logic. The contributions are structured in three parts: the model part, the enterprise integration part, and the implementation part. In the model part, we introduce the SPU container model. In the integration part, we present the integration of SPUs with business processes and enterprise applications. In the implementation part, we present the architecture and an evaluation of our SPU runtime environment.

SPU Container Model

We introduce SPUs as abstraction and encapsulation concept for event-stream processing. SPUs are containers for event stream processing tasks; they encapsulate event stream processing in a generic way that is well suitable for distributed setups and enables reuse in different contexts. SPUs demand input event streams. The granularity of this input data, i.e., of the events, is an important aspect from the software development perspective. The more fine granular the input events are, the better are the chances that events can be reused in other contexts. We thus discuss requirements engineering with respect to event streams and present tradeoffs between granularity and reusability. Since SPUs rely on an event dissemination infrastructure that influences the Quality of Service (QoS) of the whole system, we discuss event dissemination QoS with respect to event stream processing.

The contributions in the model part are:

- The SPU container model that defines structure and execution semantics of SPUs as containers for the encapsulation of event stream processing logic in manageable units;
- a requirements engineering methodology that takes event-based characteristics into account and focuses on the tradeoff between reusability and event granularity; and
- quality of service aspects of the interface between the event dissemination infrastructure and SPUs.

SPU Integration with Business Processes

SPUs can be used to build distributed stand-alone event stream processing applications. In addition, the management and scalability capabilities of the SPU container model make SPUs well suitable for the integration with (enterprise) applications. As the adoption of services in enterprise applications shows, the encapsulation of functionality in manageable units is worthwhile. In enterprise architectures, services implement business functions and the execution of whole business processes results in service interactions at the IT layer. Like services encapsulate business logic in a request/reply manner, SPUs encapsulate business logic invoked implicitly by the arrival of events. By this, SPUs complement services and enable the integration of reactive behavior in business processes.

The contributions in the enterprise integration part are:

- Requirements for the integration of event stream processing with business processes;
- extensions to Event-driven Process Chains (EPCs) and Business Process Model and Notation (BPMN) to support modeling of SPUs in abstract and technical business process models;
- a mapping of SPU-containing process models from the business process modeling to the business process execution layer;
- a mapping from the business process execution layer to SPUs at the IT infrastructure layer; and
- an integration of our EPC and BPMN extensions in Software AG ARIS and in the Software AG model-to-execute workflow.

SPU Implementation

SPUs rely on a runtime infrastructure for execution and management. Like services are executed in a SOA, SPUs are executed by a dedicated middleware. An Enterprise Service Bus (ESB) is often used to enable an asynchronous and decoupled communication between services. Similarly, a pub/sub system is a feasible infrastructure for event dissemination and provides a suitable foundation for SPUs. Thus, our middleware is built on top of a pub/sub system and provides manageable containers that represent SPUs. We refer to those containers as Event Applets, in short Eventlets; Eventlets are the technical representation, or implementation, of the abstract SPU concept. Our middleware is distributed and designed for scalability and elasticity, both being important aspects in large-scale applications. We also address privacy and heterogeneity issues that arise in a pub/sub event dissemination infrastructure.

The contributions at the implementation layer are:

- Lifecycle-managed event applets, in short Eventlets, as an implementation for SPUs;
- a runtime infrastructure for Eventlets that supports stand-alone execution as well as execution triggered by business process execution engines;
- a performance evaluation that shows the scalability benefits of Eventlets compared to native Complex Event Processing (CEP) applications and Java Enterprise applications;
- a case study that shows the benefits of Eventlets in terms of simplified software development;
- the application of a privacy-preserving pub/sub scheme to Eventlets and the evaluation of its performance; and
- the integration of an event transformation scheme to cope with heterogeneity issues in pub/sub.

1.4 Related Activities and Publications

Research Projects

Parts of this thesis are inspired by research conducted in the BMBF project ADiWa[1] [156] (Alliance Digital Product Flow) as well as in the BMBF Software-Cluster project EMERGENT[2] [76]. The goal of ADiWa was the integration of events with business processes. Industrial partners and research partners collaborated in different use cases where event processing was explored as technique to enhance business processes in the short term. In ADiWa, event streams were not directly integrated with business processes; only after the detection of complex events business functions are triggered for compensation. The discussions with industrial partners in terms of integration of event stream processing showed the demand for the seamless integration of event stream processing and motivated the concepts in this thesis. QoS of the event dissemination infrastructure was investigated in ADiWa as well. Since processes are influenced by events, a reliable event transport is required. Our work in ADiWa is the foundation for the parts of this thesis related to QoS of the event dissemination infrastructure.

One goal of the EMERGENT project is the integration of dynamic business processes across company borders. Next to services, event-based components were identified as essential to develop new added value processes. These demands in EMERGENT showed again the importance of mechanisms to seamlessly integrate event stream processing with enterprise applications.

Research Collaborations

In the context of our research on QoS we are involved with the Standard Performance Evaluation Corporation[3] (SPEC). We have collaborated with two major vendors of open source message-oriented middleware – Apache and JBoss. We have evaluated their JMS brokers[4] – ActiveMQ and HornetQ

[1] www.adiwa.net
[2] www.software-cluster.de
[3] www.spec.org
[4] http://www.spec.org/jms2007/results/

– using the jms2009-PS and SPECjms2007 benchmarks. The experience with Apache ActiveMQ and JBoss HornetQ made us confident that a reliable and high-performance SPU middleware can be based upon JMS.

Within the ADiWa and EMERGENT projects as well as within the context of Software Campus[5] research projects we have established a cooperation with Software AG[6]. Software AG also sees a high demand for the integration of event streams with enterprise software systems. As a possible solution we presented the SPUs container model as well as our concept for the integration of event stream processing with business processes by means of SPUs. We received good feedback and collaborated for a proof-of-concept implementation of our SPU modeling approach in Software AG's business process management products.

In collaboration with Purdue University[7] we addressed security and heterogeneity issues in pub/sub systems. The obtained results are used as starting point for the privacy-preserving pub/sub scheme as well as for the event transformation approach we suggest for our Eventlet implementation.

Related Publications

In the following we present our previous work in the context of this thesis. The focus of this thesis is the encapsulation and integration of event stream processing. The benefits and challenges regarding the integration of event stream processing with enterprise applications and business processes are discussed in [35, 37]; event stream processing is introduced as a paradigm to enable reactive behavior. In this thesis we introduce SPUs as containers for such event stream processing. Our SPU container model along with its implementation in form of Eventlets and the Eventlet middleware is presented in [9]; this publication is the foundation for parts of Chapters 3, 6, and 7.

In Section 6.3 we suggest a privacy-preserving implementation of Eventlets. Our approach, along with its evaluation in Section 7.2, is based upon work presented in [120, 121]. The event transformation approach (Section 6.4) that addresses heterogeneity issues in the context of Eventlets is presented in [65, 73, 74].

The integration of event stream processing with business processes by means of SPUs is presented in [10], where we focus on process modeling with BPMN. We extend this work in [11, 98] by adding support for the modeling of SPUs in EPCs. We also present an implementation for the modeling and execution of SPU-containing process models. The work on the integration of event stream processing with business processes is the foundation for Chapter 5.

We also conducted research in the area of performance evaluation of message-oriented middleware. Our SPU execution environment builds on top of such message-oriented middleware, which is used for event dissemination. We evaluated the performance of messaging infrastructures intensively. Several results have been published: jms2009-PS, an extended version of the SPECjms2007 benchmark, is presented in [150–152]. Jms2009-PS focuses on pub/sub communication and is suitable to evaluate middleware capabilities demanded by an SPU execution infrastructure. Jms2009-PS was also adapted to evaluate pub/sub in middleware that implements the Advanced Message Queuing

5 www.softwarecampus.de
6 www.softwareag.com
7 http://www.cs.purdue.edu

Protocol (AMQP) [7]; the results are presented in [14]. A workload specification that addresses specifics of event-based systems is presented in [12]. Performance evaluation is one aspect in QoS considerations for Event-based Systems. General QoS aspects within the context of Event-based Systems are discussed in [13, 35]. This work is the foundation for Section 3.3, where QoS requirements with respect to an SPU execution environment are discussed.

1.5 Structure and Terminology

Within our thesis we present abstractions for event stream processing at the conceptual layer, at business process layer, and at the technical layer. We use distinct terminology at each layer for a clear separation. At the conceptual layer we present our SPU container model for the encapsulation of event stream processing application logic. SPUs are one type of event-based component. They hold entity-centric event stream processing application logic; at runtime SPU instances are created that receive and process entity-instance-centric sub streams of events. We then apply our SPU container model to integrate event stream processing with business processes. At this business process modeling layer we introduce Event Stream Processing Services (ESPSs) and Event Stream Processing Tasks (ESPTs) that represent SPUs in EPCs and BPMN business process models. At the implementation layer we present Eventlets as the technical realization of SPUs. Eventlets implement SPU runtime semantics. Eventlets can be used for stand-alone event stream processing applications as well as in the context of business process execution where event stream processing is part of a larger application scenario.

The thesis is structured as follows: in Chapter 2 we present background information. We introduce event-based systems, pub/sub-based event dissemination, and event-processing techniques. We also give an introduction to business process modeling and execution. In Chapter 3 we introduce our SPU container model and its execution semantics. We discuss QoS aspects and show how SPUs can be interconnected. In Chapter 4 we present our requirements engineering approach that focuses on the tradeoff between event reusability and event granularity. Focus of Chapter 5 is the integration of event stream processing with business processes by means of SPUs. We present requirements for such an integration along with extensions to EPCs and BPMNs as well as an implementation. In Chapter 6 we present Eventlets - our implementation of SPUs - and our distributed Eventlet middleware. We discuss extensions to our Eventlet middleware to ensure privacy and to deal with event heterogeneity. In Chapter 7 we present the evaluation of our approach. We show the performance benefits achieved through our Eventlet middleware distribution automatisms as well as the simplification in the development of distributed event stream processing applications. We also evaluate the overhead introduced by privacy-preserving pub/sub techniques. Chapter 8 presents work related to our concept of SPUs, to our requirements engineering approach, to the integration of event stream processing with business processes as well as to our Eventlet implementation. We close our thesis with a conclusion in Chapter 9 and the discussion of future work in Chapter 10.

2 Background

In the following sections we introduce basic concepts in the area of Event-based Systems (EBSs). We present different event processing paradigms and compare event-based interaction modes with traditional interaction paradigms. We introduce publish/subscribe (pub/sub) as basic event dissemination technique. We also discuss business process modeling and execution as foundation for the integration of event stream processing with business processes.

2.1 Event-Based Systems

The event-based paradigm is well suited to integrate reactive behavior in applications. In a broad definition, an event is a significant change in the state of the universe [46]. This makes two events distinct even when time is the only thing that changes. In EBSs, events, i.e., relevant changes of state are used as active source of information. Reifications of events are created by event producers and sent to an event dissemination middleware, more specifically, to a pub/sub system. Event consumers specify interest in events in form of subscriptions. The pub/sub system matches published events with registered subscriptions and forwards events to interested event consumers. The event delivery triggers the invocation of further application logic that performs the processing of the events (implicit invocation). We use the term Event-based Component (EBC) to refer to components, which consume events and implement application logic to achieve reactive behavior; part of an EBC is the subscription, i.e., an EBC is an event consumer. EBCs can be stand-alone applications to fulfill a dedicated event-based task, e.g., monitoring of temperature. EBCs can also be integrated with existing applications and processes, e.g., with logistics processes, to add reactive behavior. An EBC encapsulates event processing logically; the technical implementation can rely on distributed and scalable event processing techniques.

In [33], Bruns and Dunkel analyze design principles for event processing systems and derive design patterns and a reference architecture. They identify three layers: the event monitoring layer, the event processing layer, and the event handling layer. Based upon this layer approach, Figure 2.1 gives a schematic overview of an EBS. In this context, EBCs are responsible for event processing as well as for event handling, i.e., the integration with other application parts. EBCs can act as event producers themselves and send events to the pub/sub system.

2.1.1 Event Dissemination with Publish/Subscribe

Pub/sub functionality is often provided by a dedicated message-oriented middleware. This middleware runs on a single broker, or on an interconnected network of brokers; event producers and consumers connect to this middleware. Event consumers register subscriptions in which they specify the events of interest; subscriptions are filter expressions and typically expressed with a query-language like syntax. Event producers send events to the middleware (publish); it is then the responsibility of the pub/sub system to match events with issued subscriptions. Different mechanisms can be applied

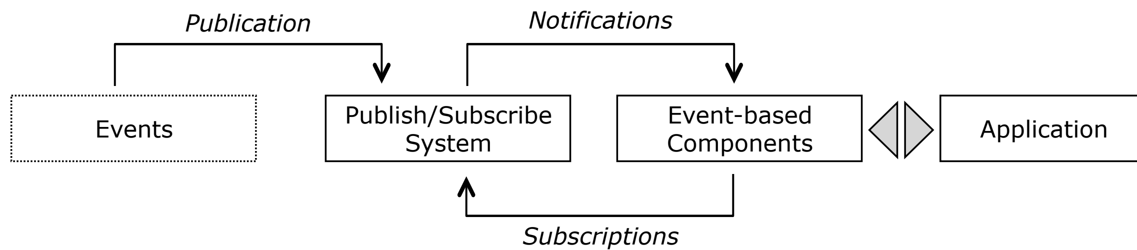


Figure 2.1: Components of an event-based system: Events are disseminated via publish/subscribe and processed by event-based components.

to establish the matching between published event and issued subscriptions [110]. Depending on the capabilities of the pub/sub system, the matching is performed on a *per-connection* basis, on a *per-event* basis, or in a hybrid approach the combines aspects of both.

When the matching between events and subscriptions is performed per connection, event producers provide metadata about the events they are going to publish, e.g., a producer will publish Champions League soccer results. This information is typically part of the connection setup between event producers and the pub/sub system. A subscription specifies the interest in a certain sort of events, e.g., Champions League soccer results. This allows a static routing between event producers and subscribers based upon the provided metadata. Examples for this type of pub/sub are channel-based pub/sub or subject-based pub/sub systems [149]. In channel-based pub/sub producers send events to a dedicated channel; subscribers register for events from this channel, e.g., Champions League Results channel. In subject-based pub/sub producers specify the subject of the events they produce whereas subjects are organized in hierarchies, e.g., `Sports.Soccer.ChampionsLeague`. Consumers subscribe for subjects; subscriptions may contain wildcards, e.g., `Sports.Soccer.*`.

When the matching between events and subscriptions is performed per event, event producers do not provide metadata about the events they will publish a priori. Subscriptions are specified on the event content and the pub/sub system evaluates subscriptions against each event submitted to the system. The matching of events with subscriptions is called content-based pub/sub [146]. Different flavors of content-based pub/sub exist. The event content can be attribute/value (att/val) based or XML-based [148]. Events can be split in header and body; in this case content-based subscriptions specify a filter on content in the event header. For an att/val event, the subscription typically has a SQL-like syntax, for XML events, the subscription uses XPath or XQuery.

Besides pure per-connection and per-event-based event routing, hybrid solutions exist. In type-based pub/sub the producer specifies the type for each event; subscriptions specify interest in certain types of events [119]. In type- & attribute-based pub/sub, subscriptions define the type of interest as well as filters on additional event content, e.g., on attributes [63, 119]. Topic-based pub/sub combines hierarchical subjects and filters on event attributes [166]. Hybrid approaches are a tradeoff between performance and flexibility; expressive subscriptions and flexibility at producer site are traded for a fixed topic per connection. An overview of the different pub/sub mechanisms is given in Table 2.1; it shows options that have to be set per connection and per event, as well as resulting subscriptions.

Subject-based Publish/Subscribe

Per Connection:	`subject := Sports.Soccer.ChampionsLeague`
Per Event:	-
Subscription:	`subject == Sports.Soccer.ChampionsLeague`

Topic-based Pub/Sub

Per Connection:	`topic := Sports.Soccer`
Per Event:	`league := ChampionsLeague`
Subscription:	`topic == Sports.Soccer && league == ChampionsLeague`

Content-based Pub/Sub with Att/Val Events

Per Connection:	-
Per Event:	`type := Sports; sports := Soccer; league := ChampionsLeague`
Subscription:	`type == Sports && sports == Soccer && league == ChampionsLeague`

Table 2.1: Publish/Subscribe mechanisms.

The tradeoffs between the different pub/sub matching mechanisms are shown in Figure 2.2. Best suited for event dissemination in EBSs is matching of publications and subscriptions on a per-event basis: this requires less global knowledge, e.g., about subject hierarchies, supports loose coupling, and allows a high flexibility. Producers decide per event about structure, type, and content. However, matching on a per-event basis is expensive in terms of event dissemination: each event needs to be inspected, evaluated against subscriptions, and routed to interested consumers.

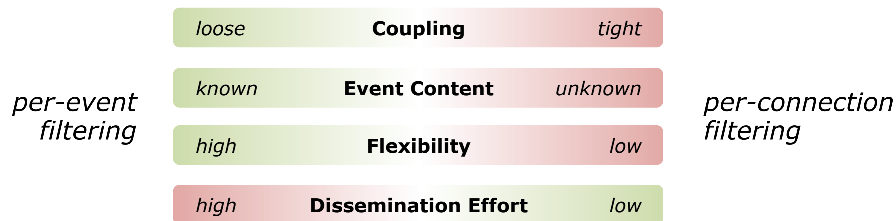


Figure 2.2: Tradeoffs between per-connection and per-event matching.

So far, all pub/sub mechanisms require global system knowledge to some extent. Event producers have to use common event types or have to be aware of the subject hierarchy of the pub/sub system. Event consumers, for example, need knowledge about event attributes on which subscriptions can be specified. This semantic coupling cannot be eliminated completely; as soon as two parties want to exchange information, a common understanding has to be established. However, mechanisms exist to relax the semantic coupling. In concept-based approaches, events and subscriptions are matched semantically, e.g., temperature events in Celsius match subscriptions in Fahrenheit [50]. In [73], we adapt this approach to characteristics of EBSs; we support event transformations inside the pub/sub system to achieve semantic-aware matching. We show in Section 6.4 how this can be applied to SPUs.

2.1.2 Publish/Subscribe Middleware

Pub/sub is an established technology and supported by various research and production-strength messaging platforms. In the industrial context, pub/sub is often a feature of Message-oriented Middleware (MOM); in addition to pub/sub, MOM typically supports queue-based communication. In the context of MOM, the term message is used to refer to the data that is exchanged between clients. Events are thus represented by messages at the technical MOM layer. The focus in industrial MOM lies on reliability and integration aspects. Since MOM is an integral part in many productive large-scale applications the availability of updates as well as advanced recovery and fail-over mechanisms are essential features. Pub/sub systems in research typically implement advanced algorithms for the matching of subscriptions with published events and focus on large distributed broker networks that can process a high volume of events.

Production-strength Pub/Sub Systems

The de-facto industry standard for asynchronous messaging is the Java Message Service (JMS) [166] where messages are exchanged via a JMS broker network. JMS brokers can be used stand-alone but are also an integral part of the Java Enterprise environment [133]. JMS defines an API only. This allows JMS applications the use of different broker implementations without modifications to the source code; only broker-specific libraries have to be exchanged. Since internals of JMS brokers are not specified, e.g., the wire level transport protocol, different broker implementations cannot inter-operate seamlessly. However, some JMS brokers, e.g., JBoss HornetQ, support JMS bridges where a broker connects as a consumer to another broker and re-distributes messages to its own clients. JMS supports one-to-one (1:1) and one-to-many (1:n) communication. One-to-one communication is implemented with queues; they decouple point-to-point communication between message producers and consumers. One-to-many communication provides pub/sub functionality and is implemented with topics. JMS provides basic content-based pub/sub capabilities via message properties. Message properties are attribute-value pairs that reside in the message header. When issuing a subscription, a consumer can use a SQL-like statement as a filter on message header attributes (message selector). The JMS broker ensures that only messages with matching properties reach the consumer. Some JMS brokers support more enhanced message selectors, e.g., AcitveMQ allows XPath expressions as message selectors that are applied to the message payload supplied in XML.

Many JMS broker vendors support the configuration of networks of brokers. Broker networks are formed to increase reliability as well as for scalability reasons. To increase reliability, fail-over mechanisms guarantee that broker failures do not result in message loss or downtime. For scalability reasons, a broker network can be used to increase the messaging capacity. Producers and consumers connect to different brokers; the broker network performs the messaging routing and ensures a reliable delivery. JMS also supports different Quality of Service (QoS) properties to control reliability aspects of the message transport: persistence, durability, and transactional behavior can be configured on a per-connection basis. Persistence controls whether messages are stored on disk prior to delivery. In case of broker failures they can be replayed upon broker restart. Durability applies to pub/sub only; it controls whether messages are cached in case of consumer failures. Without durability, consumers only receive messages during the time they are connected. With durability enabled,

the broker buffers messages in case the consumer connection is interrupted; when the connection is reestablished, the buffered messages are delivered. When transactional behavior is requested upon connection setup, several messaging operations can be grouped and executed or rolled back as an atomic unit of work. JMS is supported by a wide spectrum of products. The most popular open source brokers are Apache ActiveMQ and JBoss HornetQ. Commercial products are, for example, IBM WebSphere MQ, Oracle Weblogic, Software AG Terracotta Universal Messaging, and TIBCO Enterprise Message Service. A detailed overview can be found in [149].

Besides JMS, a messaging standard with increasing importance is the Advanced Message Queuing Protocol (AMQP). AMQP has its origin in the financial services industry. The motivation behind AMQP is the need for interoperability between MOM vendors [99, 172]. In contrast to JMS, which is an API specification, AMQP is a wire-level protocol specification. It was standardized in 2012 by the Organization for the Advancement of Structured Information Standards (OASIS) [7]. Since AMQP is a wire level protocol, different broker implementations and applications developed with different programming languages can interoperate. However, different protocol implementations can lead to different APIs at the application layer. In this case switching from one AMQP broker implementation to another requires changes to the application code. In terms of messaging semantics, AMQP provides a superset of the JMS semantics [131]; this allows the development of libraries that provide a JMS-conform API for AMQP products [14]. In the context of EBSs, a particular problem of AMQP is the lag of a standardized format for content-based pub/sub. The AMQP standard focuses on the transport layer and leaves the concrete implementation of pub/sub to the broker. Although the AMQP standardization process started with a larger scope, the final AMQP 1.0 protocol specifications does not contain messaging semantics definitions, e.g., subscription filter formats.

AMQP implementations are, for example, Red Hat Enterprise MRG, which is based upon the open source implementation Apache Qpid. Other implementations are StormMQ and VMware RabbitMQ. AMQP and JMS complement each other; AMQP ensures interoperability at the wire-level, JMS ensures interoperability at the application level. This relation is also reflected in current brokers: ActiveMQ and Terracotta Universal Messaging support AMQP in their current releases; HornetQ announced to implement AMQP support.

Another wire-level protocol is the Simple Text Oriented Messaging Protocol (STOMP) [163]. It is designed for simplicity and implements only a subset of JMS and AMQP messaging semantics. Many MOM brokers support STOMP, e.g., ActiveMQ. STOMP is mainly used in scripting languages, e.g., Perl and Python, to connect with MOM. It lacks behind the QoS capabilities of JMS and AMQP.

A messaging protocol that originates from real-time systems is the Data Distribution Service (DDS) [128]; it was standardized by the Object Management Group (OMG) in 2007. The DDS standard comprises a wire-level protocol as well as an API and a messaging semantics specification. DDS supports a fine-grained adjustment of QoS properties, e.g., bandwidth limitations and transport priorities for messages. DDS implementations are, for example, OpenDDS and PrismTech OpenSplice. Although DDS is a powerful messaging standard, the importance in enterprise applications is minor compared to JMS and AMQP.

In the area of web services, WS-Eventing [180] and WS-Notification [126] specify pub/sub communication between services. Services issue subscriptions and are notified when desired events are published. A comparison of WS-Eventing and WS-Notification with, amongst others, JMS is

presented in [91]; the convergence of both standards is discussed in [51]. In WS-Notification, subscribers can specify content-based filters as XPath expressions on the message content. Events are encoded as SOAP messages; this results in a large overhead in terms of message size and computational effort compared to JMS and AMQP.

Research Pub/Sub Systems

Pub/sub systems are also addressed in current research. While the above introduced pub/sub standards and protocols are designed for and used in productive systems, pub/sub systems in research are meant to evaluate new concepts and features. Research pub/sub systems are often distributed and support content-based pub/sub [8]; they are commonly referred to as Distributed Event-based Systems (DEBSs) [119]. Distribution is an important property to be able to cope with a high volume of events. With the spread of mobile devices, events are produced and received anywhere; this makes some sort of distributed middleware indispensable for future applications. DEBSs enable scalable pub/sub-based event dissemination. DEBSs consist of a network of brokers; the brokers are interconnected and form a topology. Event producers and subscribers connect to so called edge brokers. Events are then routed through the broker network from event producers to interested subscribers. The challenge in DEBSs is the avoidance of sending events to all brokers (flooding). To achieve this, subscriptions are merged and the resulting filters are moved as close to the event publishers as possible. This is shown in Figure 2.3; the filter for general sports events can be moved close to the event producer to reduce traffic. Parts of the filter that cannot be merged remain on the brokers close to the subscribers. To apply subscription merging and to be able to move filter expressions close to event producers efficiently, advertisements are often a prerequisite. With advertisements, producers inform the broker network about the event types they are going to publish. This allows an intelligent placement of merged subscription filters rather than flooding subscriptions or events through the broker network.

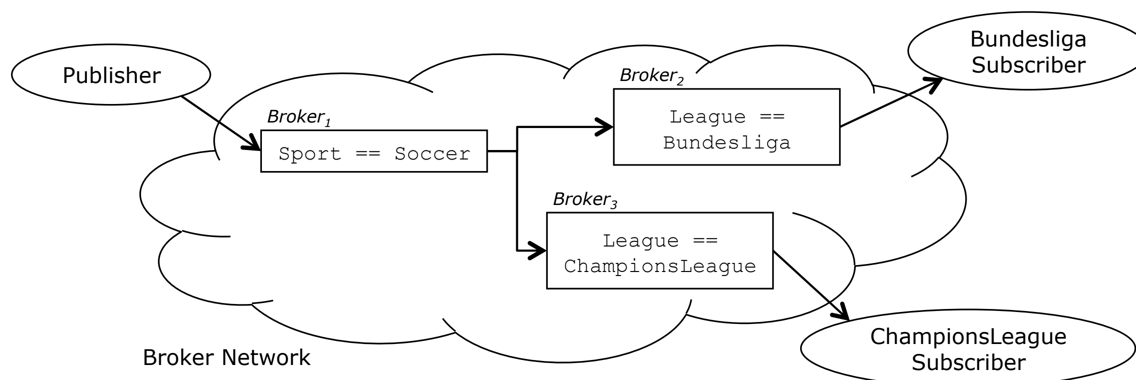


Figure 2.3: Subscription merging in distributed publish/subscribe systems.

Research prototypes of DEBSs are, amongst others, Hermes [140], PADRES [71], Rebecca [72], JEDI [54], and SIENA [40]. Prominent research topics include routing [68] and filtering mechanisms [67] in DEBSs; depending on the applied routing scheme, for example, performance is traded for false positives or false negatives in the pub/sub process. Further, the support of transactions is an important property for DEBSs in the context of enterprise applications [106]. An overview of DEBSs

that addresses, amongst others, routing, filtering and transactions is given in [36] and [141]. Some DEBSs mechanisms, e.g., sophisticated routing, are well understood in research. However, they are only rudimentarily implemented in production-strength pub/sub systems like JMS brokers.

2.1.3 Event Processing

EBCs issue subscriptions and receive matching events as input for further event processing. We distinguish between the processing of single events and the processing of streams of events. The processing of single events is stateless, i.e., each event is seen distinctly. The processing of event streams is stateful, i.e., an event can influence the processing of following events.

Processing of Single Events

In many use cases, EBCs react to single events only: in graphical user interfaces, for example, a single click event triggers application logic. In business processes single events, e.g., exceptions, trigger exception-handling procedures. Single events are used to decouple components and realize asynchronous and implicit invocations. This decoupling can happen at the technical layer as well as at the logical layer. At the technical layer, single events replace, e.g., remote procedure calls. In this case, however, the caller must still know the callee. A pure event-based interaction at the logical level does not assume such explicit knowledge about communication partners. In software artifacts this general approach is implemented by the observer pattern [78].

Processing of Event Streams

Besides reacting to single events, EBCs can perform processing of streams of events. An EBC receives events according to its subscription; these events arrive continuously as they are produced. The processing of event streams can be categorized in two main areas: event composition and pattern detection, and continuous queries over event streams [55]. An example for event composition and pattern detection is detecting fire based upon temperature and smoke events; the fire event can be seen as a complex event derived from base events. A continuous query, for example, can be used to calculate the average temperature over a continuously changing set of temperature events. EBCs can also implement event enrichment and transformations. Received events are enriched with additional information and published again. Received events can also be transformed and republished in another representation. Enrichment and transformation can be stateless as well as well as stateful.

Examples for EBCs that perform composition and pattern detection are event condition action (ECA) rules [43, 58]. Upon a certain event, an action is triggered whereas this triggering event can be a composite event. Event composition and pattern detection are also an integral part of Event-driven Architectures (EDAs) [117] and in Event-driven Service-oriented Architectures (ED-SOAs) [69, 70, 75, 105]. Components subscribe or register for certain events or event patterns and perform operations upon event arrival; events are used to trigger implicit invocations. Traditionally, the term Complex Event Processing (CEP) referred to event composition and pattern detection, while the processing of continuous queries is the application domain of Data Stream Management Systems (DSMSs). Meanwhile, both areas converge and the term CEP is often used generically to refer to both application domains. Also, many CEP engines, e.g., Esper or Software AG

Apama, support both, event composition and pattern detection, as well as continuous queries. CEP and data stream processing can be seen as complementary [42]. They have in common that they both operate on streams of events.

Event composition and pattern detection are based upon the correlation of events. Event pattern detection is the basis for event composition: when certain events have been detected, a newly composed event can be generated based upon the source events. The semantics of pattern detection depend on the underlying event algebra and consumption modes [43]. With *recent* consumption mode, for example, only the newest event instance of a certain event type is used as initiator for a pattern detection. With *chronicle* consumption mode the oldest event instance is kept as initiator; it is discarded after a successful composition. For the composition of events, point- or interval-based composition semantics can be applied [2]. With point-based composition semantics, a timestamp is assigned to the composite event, for example, the arrival time of the last event used to complete the pattern match. With interval-based composition semantics, a time interval is assigned to the composite event, for example starting with the arrival of the first event of the pattern and ending with the arrival of the last event in the pattern. Since event composition and pattern detection are applied to continuously changing streams of events, the detection of event patterns can be restricted to a time window in which events are considered valid. Further, the exact order of events that constitute a pattern can be relevant or irrelevant.

Continuous queries over event streams are similar to database queries; they are often specified in a SQL-like query language. An event stream is the equivalent to a relation in a database; single events correspond to tuples. Events arrive continuously; some relational operators, however, rely on a finite set of tuples [83]. A nested loop join, for example, is blocking when applied to an indefinite stream of events; a cross joins requires keeping all arriving events in memory. To address these issues, continuous queries are defined on so called windows. A window specifies the set of events to which a query is applied. Window definitions include information about the window size as well as about the window movement. Common window size definitions are based upon time or upon the number of events inside a window. A time-based window, for example, contains all events, which arrived within the last hour. A size-based window, for example, contains the last 100 events. The window movement can be sliding or tumbling; a time-based sliding window moves continuously as the time proceeds, a size-based sliding window moves with the arrival of each new event. Wile events are discarded continuously in sliding windows, the event set in tumbling windows changes when the defined size or time limit is hit. A size-based tumbling window, for example, moves on as soon as a defined number of new events have arrived. A time-based tumbling window, for example, moves on after a certain amount of time.

Event pattern detection, event composition, and continuous queries are supported by a variety of research and industrial software solutions. Event composition and event pattern detection, for example, have their origins in active databases where data changes or external events trigger application logic [57, 138, 139, 177]. Outside of Database Management Systems (DBMSs), business rule engines, e.g., JBoss Drools Expert, operate at the enterprise integration layer and support ECA rules. Event pattern detection, event composition, and continuous queries altogether are supported by CEP engines, e.g., research prototypes like Cayuga [29] or SASE [181] as well as production-strength CEP systems like Esper or Software AG Apama. A comprehensive survey on stream processing and

CEP along with a classification according to a unified processing model is presented in [55]. CEP as a programming paradigm became popular around the year 2000 [108]; also, the first commercial products were available at this time. Current research in the area of event stream processing focuses on scalability and performance issues.

Convergence of Event Processing and Pub/Sub

In our model of EBSs we distinguish between the event dissemination and event processing part. This logical separation is not necessarily found at the implementation layer. While systems in production environments often follow this schema, many research systems integrate event processing with the event dissemination infrastructure.

In productive systems that use event-processing technology, stateful operations on event streams are separated from the event dissemination. Events from various application components are consolidated and used as input for the EBCs, e.g., CEP engines. In a typical application, components report events using, for example, a JMS-based MOM. A CEP engine like Esper subscribes for events using JMS. This separation of concerns (event dissemination and event processing) in productive environments has been proven beneficial. It simplifies the component management: the pub/sub infrastructure and the event-processing infrastructure can be operated and maintained independently. Due to this loose coupling, event dissemination and event processing solutions of different vendors can be combined and replaced separately. However, in terms of QoS, event dissemination and event processing are interdependent. Without a sufficient event rate from the pub/sub infrastructure, for example, a CEP engine cannot guarantee a certain throughput. Thus, Service-level Agreements (SLAs) are necessary for the integration of event dissemination and event processing in enterprise applications.

In many DEBSs, pub/sub and event processing converge. Event pattern detection and filtering are moved close to the sources of events. The evaluation of continuous queries and the detection of patterns can be performed by broker nodes inside the network so that traffic is minimized. Although this distributed setup brings scalability and performance benefits, it does go at the expense of manageability. The more monolithic a software system is, the less flexible it is with respect to management and maintenance [23, 82].

2.1.4 Event-based Interaction

EBSs can be stand-alone applications, e.g., for traffic management or health monitoring [89, 158]. Event-based functionality can also be integrated with existing applications, e.g., with enterprise software systems, to add reactive behavior [76]. The integration of event-based functionality reflects an additional mode of interaction that complements existing approaches. Figure 2.4 shows the different interaction modes between participants, e.g., components, in software systems [36].

Consumer-driven interaction is common in enterprise applications. During the execution of a business process, for example, application logic is invoked explicitly to fulfill a certain task, e.g., to process a credit card payment. In cases where the exact application component that delivers functionality, e.g., a service, is known, a direct request/reply interaction applies. In cases where only the demanded functionality is known but not the concrete software component that provides

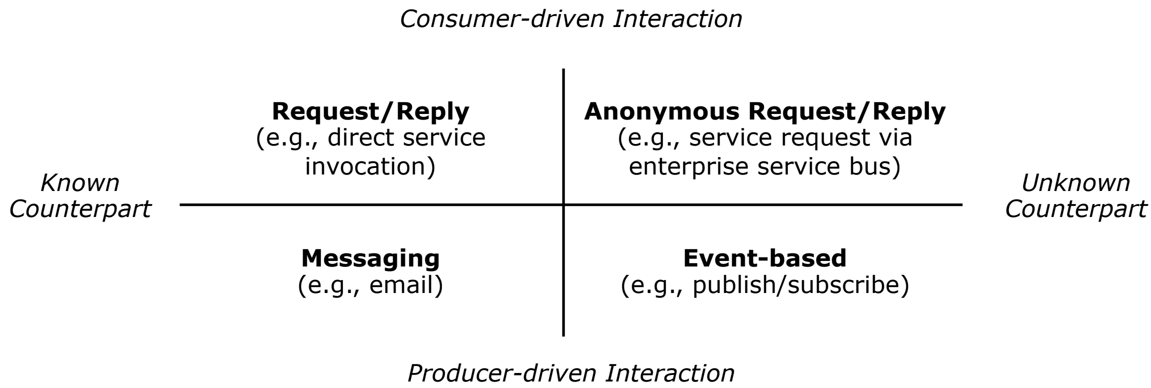


Figure 2.4: Interaction modes in distributed systems (based upon [36]).

this functionality, anonymous request/reply applies, e.g., lookup of a service in a service registry. Producer-driven interaction allows the implementation of reactive behavior. Functionality is not invoked explicitly upon demand but implicitly when available data arrives. When a data producer knows which functionality should be triggered, the counterpart is known (messaging interaction). In event-based scenarios, the reactions to events are unknown from the event producer perspective; event producers just publish events.

Reactive behavior is a natural interaction mode with the environment. Humans, for example, act or react depending on the current circumstances; they start a conversation actively (act) but react in dangerous situations. Applied to software systems, this means that active and reactive behavior should be combined to allow comprehensive integration with the environment. The foundations for the implementation of such reactive behavior are event-based interactions [47, 109, 169]. They complement request-oriented interactions in software systems as shown in Figure 2.5. Many components in software systems follow the pull-paradigm. They operate on persistent data and execute functionality upon explicit invocation. To achieve reactive behavior, data needs to be integrated in a push-based manner, like in the form of event streams. This shifts invocation to the event producers and makes it implicit. Event producers publish events; EBCs issue subscription and implement the implicitly invoked application logic.

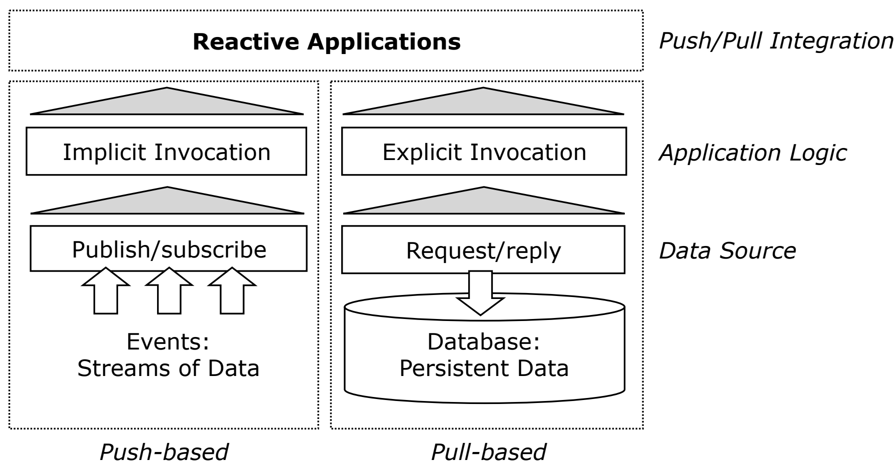


Figure 2.5: Combination of pull- and push-based interactions to realize reactive applications.

For the integration of push- and pull-based components, encapsulation of event processing application logic at the right abstraction level is a key factor for good system manageability and maintainability [23, 31]. Our Event Stream Processing Unit (SPU) container model provides such an abstraction for event stream processing. It allows the integration of push-based and pull-based functionality, for example in the context of business process execution as we will show in Chapter 5.

2.2 Business Process Modeling and Execution

Business process modeling and execution is widely adopted in enterprises. Processes are modeled by business experts and translated into executable workflow representations. They are executed inside IT infrastructures, e.g., Service-oriented Architectures (SOAs) and workflow management systems. With the adoption of the Internet of Things (IoT) and Cyber-physical Systems (CPSs), huge amounts of information become available that reflect the state of the real world. The integration of this up-to-date information with business processes gives enterprises the opportunity to implement reactivity, e.g., by monitoring their business in near real-time. This allows quick reactions on unforeseen situations as well as offering new services to customers, e.g., monitoring of environmental conditions during transport of goods and handling exceeded thresholds.

Business process models describe workflows in companies in a standardized way. They document established business procedures with the goal of making complex company structures manageable. This encompasses the business perspective as well as the IT perspective. For the modeling and execution of processes, an appropriate level of abstraction is crucial to hide irrelevant details to the process modeler. Building blocks for business process modeling, business process execution, and IT infrastructure should encapsulate business functions in a self-contained way, e.g., like services in a SOA [137]. The business process model describes interactions between these building blocks.

Two common notations for modeling business processes are the Business Process Model and Notation (BPMN) [130] and Event-driven Process Chains (EPCs) [96]. EPCs are well suited for abstract business models; BPMN allows the modeling of business processes from a more technical perspective.

Throughout this thesis, the acronym BPMN refers to the current version 2.0 of the BPMN standard if not stated otherwise. While BPMN 1.0 was mainly concerned with the graphical process representation, the BPMN 2.0 standard also includes execution semantics. Thus, BPMN 2.0 can be used as a replacement for process execution languages like Business Process Execution Language (BPEL) [127] or XML Process Definition Language (XPDL) [179]. Currently, however, BPEL is still of major importance and used as technical representation of service orchestrations; the BPMN standard also provides a mapping to BPEL.

2.2.1 Business Process Modeling with BPMN

BPMN defines three model types: collaborations, choreographies, and processes. The BPMN standard uses a patient-doctor relationship as example [130]:

- **Collaboration:** A collaboration describes the interactions between different business entities from an entity-centric point of view. A patient (business entity A) tells his or her symptoms to

a doctor (business entity B). The doctor issues a prescription to the patient, which can be used to get medicine. Both collaborate for the overall goal of health treatment.

- **Process:** A process is a step-wise description of an overall workflow, e.g., a health treatment process starts with the registration of a patient and ends with a successful treatment.
- **Choreography:** A choreography models the input to/output from participants (patient, doctor) required in each process step, e.g., the *register symptoms* process step requires input from the patient.

In the following, we concentrate on process models, as they are the basis of our integration of event stream processing with business processes. BPMN processes describe workflows (human-centric) and orchestrations (service-centric) as interactions between tasks; an example process is shown in Figure 2.6. Several task types (represented by rectangles) are supported by BPMN:

- Send and receive tasks send/receive messages to/from others tasks;
- User tasks are executed by humans, typically supported by a workflow system;
- Manual tasks require non-automated user interaction;
- Business rule tasks trigger the evaluation of a business rule to check compliance of process data;
- Service tasks are mapped to service calls; and
- Script tasks perform operations on the data directly inside the process execution engine.

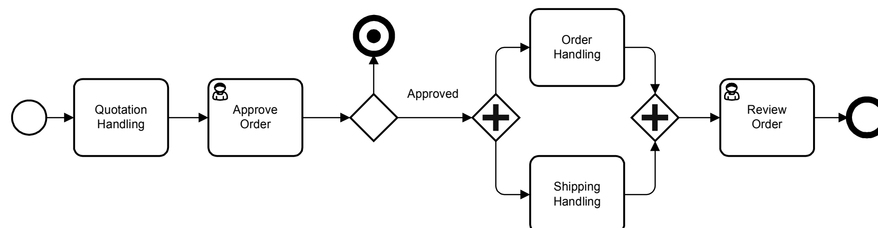


Figure 2.6: Sample BPMN process (Source: [129]).

BPMN does also specify events for asynchronous inter and intra process communication. Besides start and end events, a task can send/receive message events, exception events, signal events, and compensation events. Timer events can be used to trigger tasks periodically. Further, BPMN process models support the notion of sub processes to simplify the illustration of large processes. Process models can also contain pools and lanes to group tasks with respect to business entities whereas each entity can account for multiple tasks.

2.2.2 Business Process Execution

The implementation of business processes in enterprises involves three layers: the modeling layer, the execution layer, and the IT infrastructure layer (see Figure 2.7). During design time, business experts create models from an abstract business perspective, e.g., using EPCs [96, 170] or BPMN. These models are then refined into technical process models, e.g., using BPMN. The models are then

transformed into executable workflows represented in, e.g., BPEL. Typically, the workflow execution requires IT support, which is provided by a SOA and workflow management systems.

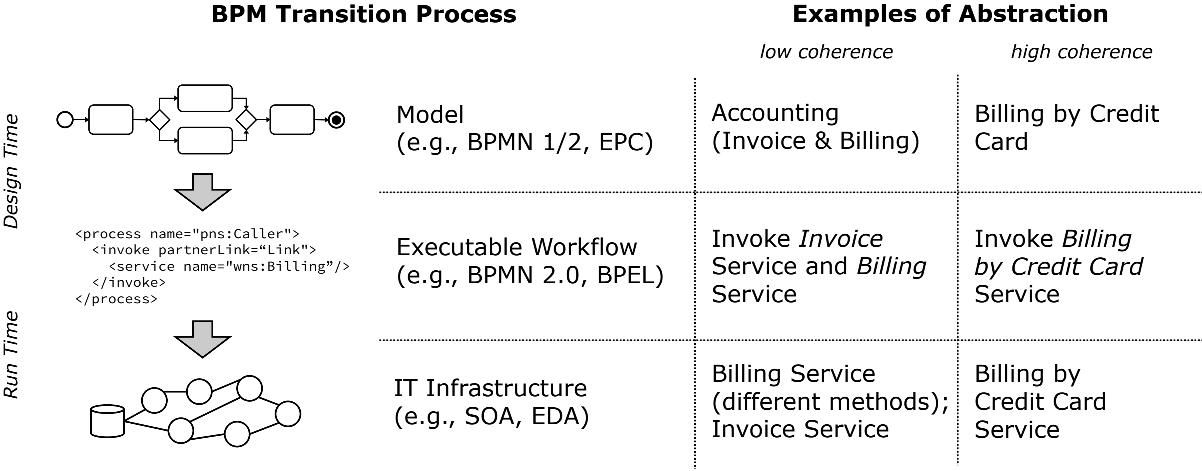


Figure 2.7: Transition steps between process modeling, process execution, and IT infrastructure layer.

The model-to-execution workflow is an integral part of the overall Business Process Management (BPM) cycle [1]. The transition process, however, from a business process model to, e.g., SOA service interactions is not trivial and requires expertise from the business perspective as well as from the IT perspective. To enable the seamless implementation of modeled processes, the abstraction of business functions should have the same granularity at each layer; a coherent abstraction across the layers minimizes the transition effort [3, 134]. The example in Figure 2.7 illustrates this: the *low coherence* case requires a refinement with each transition step (a single BPMN task maps to multiple services) while the *high coherence* case allows a one-to-one transition between the business function representations available at each layer (e.g., BPMN tasks, BPEL invocations, and SOA services).

Technical representations of business process models are used as input for process execution engines like JBoss jBPM, Activiti, or the Software AG webMethods BPMS. In BPMN, processes begin triggered by start events. The control flow proceeds from this start event to tasks, gateways, or signal elements. When the control flow reaches a task, the task is activated; it then executes and completes. BPMN uses the notion of tokens that are passed to tasks and that move on after completion. Different gateway types exist to influence the control flow: the control flow can move on in parallel to different tasks as shown in Figure 2.6 (several tokens are created) or move on a certain path depending on specified conditions. With their activation, the different task types of a process model need to be executed by an IT infrastructure. Process execution engines manage the control flow and coordinate the interactions with IT systems that provide the functionality to execute tasks. Two dominant task types in this setting are service tasks and human tasks. Service tasks are typically mapped to SOA service calls and human tasks are mapped to a workflow engine. Executable process representations provide constructs to specify these mappings. A service task, for example, requires the specification of the service endpoint along with the input data and output data.

To allow a high coherence between the business process modeling layer, the execution layer, and the IT infrastructure layer, our SPU container model provides a suitable encapsulation mechanism. SPUs allow the integration of event stream processing logic with high coherence across the layers.

3 Event Stream Processing Units

To cope with the increasing complexity of modern software systems, software engineers developed several programming paradigms and abstractions over the past decades. These abstractions map real-world concepts to software concepts. For example, Object-oriented Programming (OOP) maps real-world entities (e.g., rooms, persons, or invoices, etc.) to software entities [93]. The requirement for scalable and distributed infrastructures led to further abstractions like service-oriented architectures (SOA). SOAs are based on the abstraction primitive of services. A service encapsulates a well-defined functionality and can be invoked by an explicit request. Services are managed by a middleware infrastructure that provides for naming, locating, invoking, and persisting of services.

The pervasive use of mobile devices and sensors makes the event-based approach an indispensable software development paradigm. This has led to a new generation of sense and response systems [45]. Patient monitoring, smart home environments as well as traffic management systems are typical applications that use an Event-driven Architecture (EDA) [52, 86, 89]. These Event-based Systems (EBSs) lack a crisp equivalent of the notion of a service. Services encapsulate application logic and are an abstraction mechanism for developers to simplify the realization of distributed and scalable infrastructures. In this chapter, we present a comparable abstraction mechanism for event stream processing. We introduce Event Stream Processing Units (SPUs) as containers for event stream processing application logic. SPUs are the necessary building blocks for modern reactive applications. They are designed with respect to the push-based nature of event streams. They encapsulate event stream processing in an entity-centric way and enable distributed and scalable event stream processing applications.

Our SPU container model relies upon a pub/sub infrastructure for event dissemination; we introduce the notion of an event bus as pub/sub communication layer. Quality of Service (QoS) is essential for the integration of SPUs with other components; we thus discuss QoS in the context of event dissemination and event processing. We further present a use case for event stream processing with a network of SPUs.

3.1 Event Bus

The pub/sub system is the communication layer in EBSs [132]; it is responsible for the event dissemination. We refer to this communication layer as *event bus*. Our approach of SPUs does build on top of this communication layer. The concept of SPUs is agnostic to the concrete implementation and technical details of the pub/sub provider; we only require a general pub/sub interface to be exposed. In the following we define such a generic interface to a pub/sub system as foundation for SPUs.

As introduced in Section 2.1.1, a subscription defines a filter for events of interest. For SPUs we assume that this filter is specified upon the content of events. The filter is stateless, i.e., it can be applied to events independently to determine whether they meet the filter conditions. No additional knowledge other than the filter and the event is necessary for this evaluation. Subscriptions are the

basic mechanism for SPUs to express their interest in events. The publication of an event results in a notification as reification of the event inside the pub/sub system. The pub/sub system evaluates filter conditions specified in subscriptions against notifications to determine whether a certain notification matches a subscription. As counterpart of subscriptions, event publishers can specify advertisements. An advertisement specifies the types of events to expect from a publisher. Advertisements are not required for SPUs, however they allow plausibility checks, i.e., when an SPU issues a subscription it can be checked whether events of the demanded type are already provided by publishers.

Pub/sub-based event dissemination requires common knowledge between event producers and subscribers about the content and the structure of events; even content-based subscription must be specified in accordance to the event payload. This problem is generic to distributed systems; it is especially problematic in EBSs since producers and consumers are decoupled conceptually and semantically. One solution is an implicit definition of event types, i.e., events are self-describing [132]. Another solution are transformation approaches inside the middleware; rather than sharing knowledge between all event producers and subscribers, producers and subscribers need to establish a common knowledge with an intermediate party only. In Section 6.4 we will present an application of our middleware-based transformation approach, presented in [73], with respect to SPUs.

Common Publish/Subscribe API

SPUs rely on a generic API for pub/sub [141]. Event publishers send events to the pub/sub system using the publish method. No further data other than the event is required so that producers do not require additional knowledge. Prior to publishing events, producers can advertise the type of events they will later publish; this is optional in the context of SPUs, however required by numerous pub/sub systems:

```
advertise(EventType t);
```

Publishing events of advertised types is now possible asynchronously without the need for further parameter exchange with the pub/sub system:

```
publish(Event e); // Event e is of type t
```

Interested consumers subscribe for events by specifying a filter on the event content:

```
subscribe(Filter f);
```

The pub/sub system evaluates the subscription filter f against the published events e. Upon a match, the event e is delivered to the interested subscribers, i.e., a method at the subscriber is called that handles the incoming events:

```
onEvent(Event e); // Method is called upon arrival of Event e that
   matches filter f
```

The above introduced generic pub/sub methods can be mapped to existing pub/sub systems. We developed a translation layer for the Java Message Service (JMS) (see Section 6.2) to implement these generic pub/sub calls. Other productive and research pub/sub systems support these basic constructs accordingly.

The goal of an abstraction mechanism is to provide an intuitively understandable way to group functionality. For services, this is business functions, i.e., a service encapsulates functionality that can be understood from the business perspective. Domain experts without in-depth IT knowledge should be able to understand what a service does. We follow the same goal with SPUs. SPUs are containers for generic reactive application logic referred to as *generic tasks*. Typically, such a task represents functionality associated with a certain entity type, e.g., a shipment. Further, this functionality relies on input in the form of streams of events from a pub/sub system. A single SPU instance receives the event stream associated with a certain entity instance, e.g., events for a particular shipment.

As introduced in our shipment monitoring use case in Section 1.2, SPUs can be used to encapsulate shipment monitoring application logic in an entity-instance-centric way, i.e., an SPU instance is responsible for the monitoring of a certain shipment. The shipment monitoring application logic applies to each shipment; it is thus generic with respect to the entity type *shipment*. Figure 3.1(a) shows that such a generic task, like shipment monitoring, is defined by actions that are applied to different entity instances. For each individual entity instance, a task instance is maintained; actions are generic and automatically applied per entity instance. Figure 3.1(b) shows this for our example of shipment monitoring.

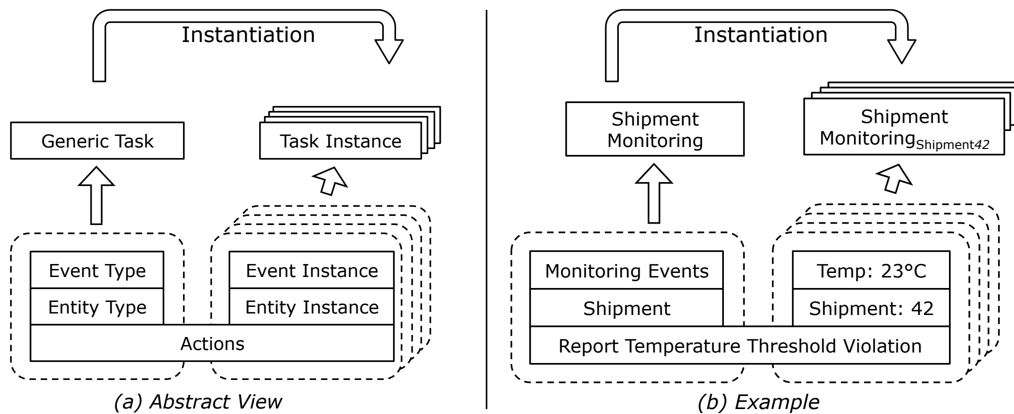


Figure 3.1: Generic task model: Abstract view (a) and shipment monitoring example (b). In (b), an SPU instance exists per shipment.

In addition to the shipment-monitoring scenario introduced in Section 1.2, we introduce a traffic control system and a health monitoring scenario to illustrate further applications of our SPU container model.

Scenario: Traffic Control System

Managing the increasing volume of traffic in large cities is a challenging task. To avoid gridlock and to share infrastructure-costs for highways or tunnels, cities rely on toll systems to charge vehicles entering certain areas. Vehicles are registered when they enter or exit certain zones and cities often have different rates for different areas, times or kinds of traffic. An IT system for traffic management

has to keep track of each car, its status, and location. Norway's AutoPASS system, for example, handles more than 2.1 million entities across Norway[1].

The process of toll collection consists of a dynamic sensing part and a standardized processing part: Cars have to be detected and tracked (sensing), toll rates and billing information have to be correlated with the detected movements (processing). These toll systems are often realized as Cyber-physical System (CPS) for the sensing and a Service-oriented Architecture (SOA) for the processing part. Vehicles are continuously detected at checkpoints throughout the city. The detections are represented by event streams, which are then used for toll processing.

While traditional event stream processing techniques can handle high volumes of events, they lack an intuitive abstraction mechanism designed for scalability on the architectural level that enables easy integration with SOAs. With SPUs however, this integration is intuitive and the resulting application is inherently scalable; SPU instances represent vehicles and can be assigned to arbitrary machines for execution. The application logic is the same for each vehicle: detection events have to be recorded and toll has to be calculated. This generic task can be modeled with an SPU as shown in Figure 3.1. For each vehicle crossing the city limits an SPU instance is created automatically. During the instantiation further information, such as the toll category for the corresponding license plate, is retrieved by a SOA service invocation. Each SPU instance receives all detection events for its corresponding vehicle only. When a vehicle leaves the city the associated SPU instance performs the billing by invoking the invoice service and shuts itself down.

The traffic control task is generic and is instantiated per car that is detected. The events of interest are position events of cars. As soon as a car leaves the city, i.e., the position indicates that the car is outside the city boundaries, an invoice is created to request the payment of toll.

Scenario: Health Monitoring

Continuous health monitoring of patients is another use case for sense and response applications. Today's sensor technology allows equipping persons with multiple lightweight sensors to collect health data, e.g., heartbeat rate or blood pressure [157]. Smartphones can then be used to collect these sensor data for long term analyses or to detect critical situations. One challenge is now the task of integrating the patients' monitoring system with the IT systems inside a hospital. Here, SPUs can help to improve the workflow inside hospitals but also to increase patient safety.

SPUs in a hospital contain patient centric application logic, with patients being the entities of interest inside a hospital. A typical scenario is a new patient entering the hospital. When this patient is equipped with the above-mentioned monitoring system the smartphone can now be used to forward all sensor data: the data is enriched with a unique patient ID and forwarded to the hospital event dissemination infrastructure. An SPU middleware now creates an SPU instance for each patient that is new inside the hospital. This SPU instance can perform patient monitoring by evaluating the events received from the patient's sensors. But also more complex functionality is possible: at creation of an SPU instance the patient database of the hospital can be queried to retrieve additional information and to trigger further administrative tasks. In general, tasks that are generic for all patients can be encapsulated into an SPU which is then automatically instantiated per patient. An

[1] http://www.q-free.com/solutions/we-did-oslo/

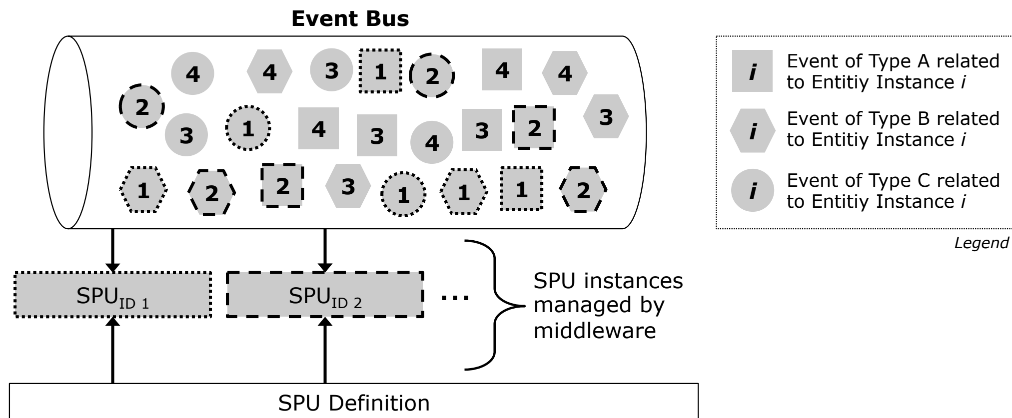


Figure 3.2: Event Stream Processing Units (SPUs) operate on entity-centric event streams; an entity-centric event stream can consist of different event types.

SPU instance can be seen as event-driven application dedicated to a patient, e.g., to monitor a patient, to archive sensor data, or to interact with legacy hospital systems. Since SPU instances are stateful and run independently, a runtime configuration, e.g., adjusting sensor threshold values, can be realized easily.

3.2.1 SPUs as Abstraction to Event Stream Processing

SPUs add a layer of abstraction on top of existing event stream processing approaches. They allow specifying event stream processing application logic on a generic basis. Rather than specifying event stream processing for multiple entity types (coarse grain) or per entity instance (fine grain), SPUs allow event stream processing to be specified per entity type. This follows the OOP paradigm where objects reflect entities. The encapsulation of event stream processing is shown in Figure 3.2: Events of different types and different entity instances are published to the event bus. SPU instances now subscribe for all events related to a single instance and perform entity-instance centric event stream processing; the creation of SPU instances and the SPU instance specific subscriptions are managed by a middleware.

The abstraction step from entity instance-centric to entity type-centric event stream processing requires concepts that address the push-based specifics of EBSs. As described in Section 2.1.1, EBSs in general, and event stream processing in particular, rely on a pub/sub system for event dissemination. A generic interface to pub/sub is the foundation for SPUs. Further, event-based execution semantics, i.e., implicit instantiation as well as implicit completion strategies need to be supported by SPUs.

3.2.2 SPU Instantiation and Completion Strategies

In pull-based environments, e.g., traditional SOAs, interactions are triggered explicitly; they are consumer-driven. Services, for example, are invoked explicitly (see Section 2.1.4). In push-based environments, interactions are triggered implicitly; they are producer-driven. Event stream processing is triggered by the arrival of events; SPUs, as container model suited for the push-based approach, need to support such implicitly-triggered interactions.

SPUs contain entity-centric application logic. At runtime, SPU instances have to be created for the processing of entity-instance related events. In typical OOP, this instantiation is explicit (creation of an object). However, the push-based nature of event streams allows implicit instantiation as well. Thus, SPUs support explicitly-triggered as well as implicitly-triggered instantiation. The same holds for the completion of event-driven tasks implemented with SPUs. Since an event-based interaction does not follow a request/reply scheme (events can arrive continuously), the completion of event stream processing needs to be triggered. The completion can either be triggered externally or internally. We refer to externally triggered completion as *explicit completion*; an SPU instance is stopped explicitly, e.g., via an external command. Completion of an SPU can also be triggered internally, from inside the SPU itself; we refer this as *implicit completion*.

Explicit and Implicit Instantiation

SPUs can be instantiated explicitly, i.e., the instantiation is controlled externally. In the example of a shipment monitoring SPU, SPU instances are created explicitly for each shipment by an external controller, e.g., an SPU instance is created for monitoring of Shipment No. 42. In the case of explicit instantiation the existence of events for a certain entity instance, e.g., for Shipment No. 42, is not a prerequisite. An SPU might be instantiated although no events are available for processing yet.

When SPUs are instantiated implicitly the instantiation is triggered based upon arriving events. The instantiation condition is specified as part of the SPU; no external trigger for the instantiation is required. An SPU instance is created, for example, as soon as events for a certain entity instance, e.g., for Shipment No. 42, arrive. An SPU runtime environment ensures, that only one SPU instance is created per entity instance. Conceptually, an SPU can also be instantiated based upon complex events, e.g., by a derived event that confirms that Shipment No. 42 has left the logistics hub. A corresponding SPU instance is then created and receives all monitoring events for this particular shipment.

Explicit and Implicit Completion

Explicit completion follows the same semantics as explicit instantiation. Active SPU instances can be shut down via external commands, e.g., a command triggering the shutdown of the SPU instance that monitors Shipment No. 42. This shutdown is independent of the events arriving for a certain entity instance.

The implicit completion of an SPU instance is triggered from inside the SPU instance. It can be based upon a condition that is evaluated continuously. Examples are timeouts or dedicated events, e.g., when no new events arrive for Shipment No. 42 before the timeout.

3.2.3 SPU Definition and Structure

SPUs encapsulate event stream processing application logic with the goal of simplifying development and integration of event stream processing components.

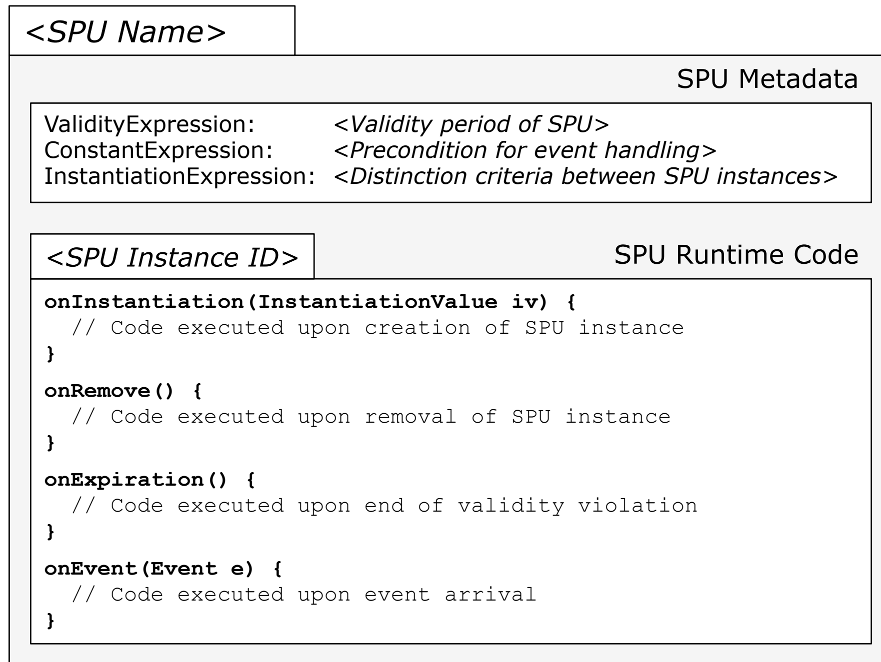


Figure 3.3: Structure of an Event Stream Processing Unit (SPU) with partitioning in runtime code and metadata required for the instantiation.

Definition: Event Stream Processing Units (SPUs)

Event Stream Processing Units (SPUs) *are containers for generic event-driven tasks. They hold application logic with respect to entity types; an SPU instance is created for the processing of entity-instance-centric streams of events. Instantiation and completion of SPUs can be triggered implicitly or explicitly. SPUs have a managed lifecycle.*

SPUs hold application logic that applies to entity types. At runtime, SPU instances process events related to certain entity instances; thus, SPU instances have to issue subscription dynamically upon their creation.

In this section, we introduce the SPU container model from a conceptual perspective. We present the technical and implementation perspective in Chapter 6. The container model describes the structure of SPUs relevant to developers. The technical and implementation perspective presents the middleware necessary to execute, distribute, and manage SPUs. To support implicit instantiation and completion, SPUs contain dedicated metadata that specifies when to create and shutdown SPU instances. Further, SPUs follow a certain structure to allow lifecycle management. The basic structure and methods are shown in Figure 3.3; we distinguish between SPU metadata and SPU runtime code. SPUs are identified by a unique name. Further, to identify SPU instances at runtime, an ID per instance is assigned by the middleware. In the following we describe the SPU metadata and runtime code.

SPU Metadata

The SPU metadata is shared amongst all SPU instances. It defines the event stream, which arrives at each SPU instance, i.e., it partitions an event stream with respect to entity instances. The SPU metadata is required for the instantiation and management of SPUs by a runtime environment. The metadata consists of three parts, the:

- Constant expression;
- Instantiation expression; and
- Validity expression.

Based upon constant and instantiation expressions, subscriptions are derived dynamically as filters for entity-instance-centric events.

Constant Expression

The event bus provides asynchronous communication in EBSs. It transports event notifications from different event producers to interested subscribers. An SPU might be interested only in certain types of events; the constant expression is a precondition to identify events relevant for an SPU and all its instances. The constant expression can be seen as a filter applied at subscription level; if an event does not match, no further processing is required. We chose the term *constant* to indicate that the resulting subscription is equal for all SPU instances.

When events are represented with XML, an example for a constant expression in the style of XPath is: `/event/type[. = positionEvent]`; this expression enforces that all events arriving at SPU instances are position events. When events are represented with attribute/value (att/val) pairs, an example for the constant expression is: `(eventType == positionEvent) AND (sensorType == A)`; this expression enforces that all events arriving at SPU instances are position events originating from sensor type A.

Instantiation Expression

The instantiation expression specifies the distinction criterion between SPU instances. Each SPU instance receives a sub stream of events associated with a certain entity instance. The partitioning into sub streams is based upon grouping attributes; an event sub stream corresponds to a certain grouping attribute value. A grouping attribute is, for example, a shipment ID. At runtime, each SPU instance issues a subscription for certain grouping attribute values, e.g., a subscription to events with Shipment ID No. 42.

When events are represented with XML, an example for an instantiation expression in the style of XPath is: `/event/shipmentID`; based upon this instantiation expression, SPU instances are created for specific shipments. The instantiation expression along with a specific instantiation value, e.g., Shipment No. 42, allows SPU instances to derive an entity-instance-centric subscription, i.e., a subscription for events of Shipment No. 42. For att/val-based events, the basic instantiation expression

contains the name of a single attribute suitable for grouping, e.g., shipment ID. In addition, instantiation expressions can specify an attribute set, e.g., when a shipment ID is not unique. An example is `(shipmentID;truckID)`. In case a shipment ID is only unique in combination with a truck ID, this instantiation expression ensures that SPU instances are created for each unique combination of shipment ID and truck ID attribute values, e.g., Shipment No. 42 in Truck No. 37.

Validity Expression

In EBSs event producers and event subscribers are decoupled and do not follow request/reply semantics. Thus, the validity expression specifies a condition for implicit completion. The validity expression is evaluated continuously throughout the life span of SPU instances. Examples for validity expressions are:

- *Timeout:* Events may arrive continuously; however, the arrival of events may also end abruptly without prior notification. A timeout occurs when no events arrive for a certain time interval;
- *Event rate insufficient:* The number of events per time interval is too low;
- *Expiration time/date:* SPU expires at a certain point in time; and
- *Expiration event:* Expiration is triggered by a dedicated event, e.g., when an event producer leaves the system.

SPU Runtime Code

Along with the metadata, an SPU has to implement four methods: `onInstantiation`, `onRemove`, `onExpiration`, and `onEvent`. These methods contain the application logic for the SPU lifecycle management and for the handling of incoming events.

onInstantiation Method

The `onInstantiation` method contains the code, which is executed when an SPU instance is created. There is no need to write any instantiation code; whether it is needed depends on the use case and the demanded SPU functionality. An example for instantiation code is opening a database connection to retrieve additional data, e.g., the temperature threshold value for a certain shipment, or registering a complex event processing query with a Complex Event Processing (CEP) engine.

onRemove Method

The `onRemove` method is executed when an SPU instance is completed - either implicitly or explicitly. It is responsible for a clean shutdown with respect to the SPU application logic. Examples are the shutdown of connections to external components, or persisting data for later reuse.

onExpiration Method

Part of the SPU metadata is the validity expression. If an SPU instance expires according to this expression, the `onExpiration` method is executed. The methods can be used by developers to react appropriately to expirations.

onEvent Method

The core functionality of an SPU is represented by the event processing application logic. The first step for event processing is the handling of incoming events; the corresponding code is located in the `onEvent` method. It is called upon the arrival of events that match the issued subscription.

3.2.4 SPU Execution Semantics

The metadata of SPUs determines the events each SPU instance receives at runtime. When an SPU instance is created (implicitly or explicitly), it issues a subscription to receive events related to one entity instance. In this section we formally describe the underlying SPU execution semantics. For this, we assume events represented with att/val pairs and values to be either strings or integers. We introduce event types in our system model and define the relation between events and SPU instances.

Events e are sets of att/val pairs where an attribute a is identified by its name n. Attribute values v are either `STRING` or `INT` values:

$$a := \{(n, v) \mid n := \texttt{STRING} \wedge (v := \texttt{STRING} \vee v := \texttt{INT})\} \tag{3.1}$$

For better readability we introduce the notation $a.n$ for the name of an attribute a and $a.v$ for the value of an attribute a.

$$a.n := \{n \mid a = (n, v)\} \tag{3.2}$$

$$a.v := \{v \mid a = (n, v)\} \tag{3.3}$$

Each event e is of a certain type t. A type t is defined as set of attribute names $a.n$:

$$t := \{a_i.n, ..., a_j.n\} \tag{3.4}$$

An event e of type t, short $e[t]$, contains all att/val pairs defined in type t:

$$e[t] := \{a \mid \forall i : a_i.n \in t\} \tag{3.5}$$

We refer to an attribute of an event $e[t]$ as $e[t].a$:

$$e[t].a := \{a \mid a \in e[t]\} \tag{3.6}$$

SPUs implement entity-type-centric event stream processing. The events that belong to an entity can be of different types, e.g., when the entity is a shipment, events can be position events and temperature events. These events of different types have to be groupable, i.e., they need to have attributes in common to derive their associated SPU instance. For example, position and temperature events can both contain an attribute that holds the shipment number. The candidate attributes $g_{t_i,...,t_j}$ for a grouping across event types $t_i, ..., t_j$ are defined as follows:

$$g_{t_i,...,t_j} := \{a.n \mid \forall k : t_k \in t_i, ..., t_j \land a.n \in t_k\} \tag{3.7}$$

The instantiation expression now defines the attribute names to partition the event stream into SPU instance centric sub streams of events. In case of multiple event types, these attributes must be present in all events, i.e., they must be part of the grouping set $g_{t_i,...,t_j}$. The instantiation expression $s^v_{t_i,...,t_j}$ of an SPU v is a set of attribute names $a.n$:

$$s^v_{t_i,...,t_j} := \{a.n \mid a.n \in g_{t_i,...,t_j}\} \tag{3.8}$$

In the basic case, the instantiation expression is a single attribute name, e.g., shipment ID. However, instantiation expressions consisting of multiple attributes are possible. This is similar to primary keys in databases; these can also consist of multiple fields to make a unique identifier. For example, when shipment IDs are only unique in combination with truck IDs, the instantiation expression contains the attributes shipment ID and truck ID. At runtime, SPU instances I are created based upon the instantiation expression $s^v_{t_i,...,t_j}$. In case of a single instantiation attribute, a specific SPU instance receives all events that have the same instantiation attribute value val. In case of multiple instantiation attributes this holds analogously; an SPU instance $I[s^v_{t_i,...,t_j}][val]$ receives all the events where the values val of the attributes in the instantiation attribute set are identical:

$$val := \{a \mid a.n \in s^v_{t_i,...,t_j}\} \tag{3.9}$$

$$\begin{aligned} I[s^v_{t_i,...,t_j}][val] := \{e_n[t_k] \mid \forall n : t_k \in t_i, ..., t_j \land \\ e_n[t_k].a \supseteq val\} \end{aligned} \tag{3.10}$$

SPU instances $I[s^v_{t_i,...,t_j}][val]$ exist for all instantiation attribute values. Each SPU instance receives the events according to the partitioning specified by the instantiation expression $s^v_{t_i,...,t_j}$:

$$\forall i : e_i[t_j].a \in s^v_{t_i,...,t_j} \rightarrow \exists I[s^v_{t_i,...,t_j}][val] \mid e_i[t_j].a \supseteq val \tag{3.11}$$

The above presented execution semantics focus on the instantiation expression. The constant expression is part of an SPU as well. In the context of the formal model, the constant expression defines conditions on arbitrary attribute values. Events are only delivered to SPU instances when the defined conditions match with the event. The subscription an SPU instance issues at its creation can be derived based on the formally described executions semantics. One SPU instance exists for each value of the instantiation attribute, respectively for each value combination in case of multiple in-

stantiation attributes. This requires a dynamically derived subscription for each SPU instance based upon the constant expression, the instantiation expression, and the values of attributes specified in the instantiation expression.

The subscription filter starts with the constant expression as static part. It then contains an equality check based upon the attributes specified in the instantiation expression: each SPU instance subscribes to events corresponding to a unique value of the attribute specified in the instantiation expression. In case of multiple attributes in the instantiation expression, each SPU instance subscribes to events corresponding to a unique value set of the attributes specified in the instantiation expression. With implicit instantiation, the runtime environment ensures that an SPU instance is created upon demand, i.e., when new entity-centric events are published. With explicit instantiation, a value set of the attributes specified in the instantiation expression needs to be specified manually upon instantiation, e.g., it is known that a monitoring SPU for Shipment No. 42 should be created. We illustrate this using the introduced shipment-monitoring scenario; we assume att/val events with the following attributes: `eventType`, `shipmentType`, `shipmentID`, `truckID`, `temperature`. The constant expression specifies the interest in temperature events for transport of food or drugs:

```
1 (eventType == temperatureEvent) AND
2 (shipmentType == food OR shipmentType == drugs)
```

The instantiation expression specifies that the event stream per shipment is defined uniquely by a truck ID in combination with a shipment ID:

```
1 (truckID; shipmentID)
```

The derived subscription filter for an SPU instance that processes events for a shipment with ID 42 inside of a truck with ID 37 is:

```
1 (eventType == temperatureEvent) AND
2 (shipmentType == food OR shipmentType == drugs) AND
3 (shipmentID == 42) AND
4 (truckID == 37)
```

Upon creation of an SPU instance, this filter is used to subscribe for events. Lines 1 and 2 are the static part of the subscription that is equal for all SPU instances; lines 3 and 4 are the dynamic part of the subscription that differs between SPU instances.

Besides the subscription derived based upon constant and instantiation expression an SPU runtime environment might issue subscriptions to receive events specified as expiration event within the validity expression. The derivation of such a subscription depends on the realization of validity expression support in an runtime environment implementation.

3.3 Quality of Service Considerations

SPUs - as an abstraction concept for event stream processing - are reactive building blocks in software systems. In this context, QoS is an important aspect to ensure proper functionality. QoS with respect to SPUs, however, is complex to define and to guarantee. Thus we discuss QoS properties for SPUs in the following sections.

Like services interact with, e.g., a service registry and an enterprise service bus, SPUs rely on a pub/sub system for event dissemination. Thus, QoS of SPUs depends on multiple system components. A schematic view on the different layers of QoS in an SPU infrastructure is shown in Figure 3.4. Events are published by producers where QoS and Quality of Information (QoI) influence the QoS of the overall system. QoS at this layer is for example the event rate at which producers publish events. QoI is related to the data itself, e.g., determined by the accuracy of sensors.

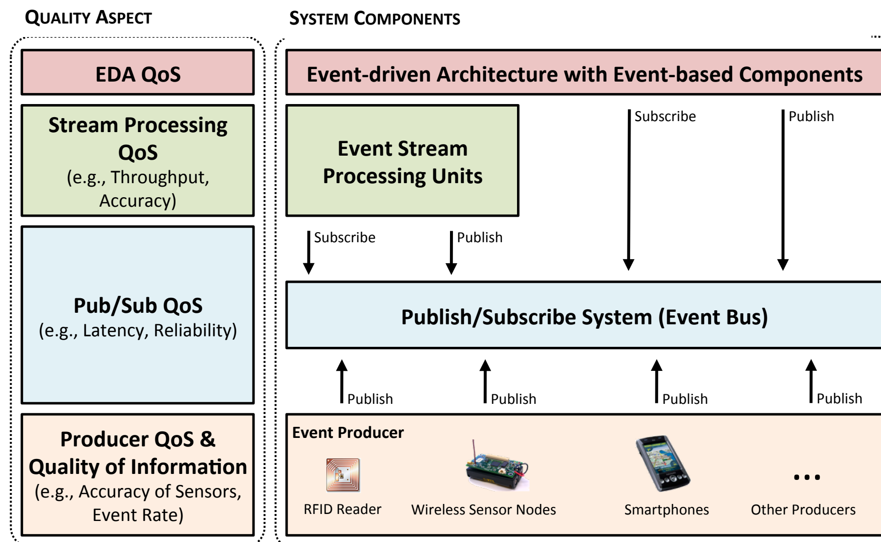


Figure 3.4: Quality of service and quality of information in event-based systems depend on event production, on event dissemination, and on event processing.

The next layer is the event dissemination infrastructure, in our case a pub/sub system. The pub/sub system is essential to disseminate event notifications in an asynchronous and decoupled way. SPUs subscribe to events and a single event notification can trigger or change the execution of application logic. It is thus necessary to make SPUs, and event consuming and producing components in general, aware of the QoS of the underlying notification mechanisms. Different QoS properties are adopted in current notification middleware, e.g., in JMS brokers [150], in the data dissemination service (DDS) [128], or in research pub/sub systems [22]:

- *Persistence*: The pub/sub middleware takes extra care to ensure that no event notifications are lost in case of a server crash by buffering them on persistent storage.
- *Delivery Mode*: The delivery mode determines whether events are delivered at least once, at most once, or exactly once.
- *Durability*: With non-durable subscriptions a subscriber will only receive notifications that are published while he is active. With durable subscriptions, notifications are buffered in case subscribers temporarily disconnect.
- *Transactions*: A pub/sub session can be transactional or non-transactional. A transaction is a set of publish or event arrival handling operations that is executed as an atomic unit of work.
- *Order*: When order of event notifications is guaranteed, the middleware ensures that notifications arrive in the order they were published.

- *Performance*: The fraction of event notifications that can be handled by the middleware within time limits (throughput and latency).
- *Security*: Guarantees by the middleware to ensure confidentiality, privacy, and integrity of notifications.

These QoS parameters apply to Message-oriented Middleware (MOM) in general. In our work, we focus on pub/sub communication. Thus, the QoS properties are discussed under event-based specifics typical to our scenarios:

- *Persistence and Reliability:* In some scenarios persistence or reliable event delivery is not required due to the transient nature of events. Temperature readings or position events, for example, might only be relevant when delivered timely. Other scenarios might require guaranteed event delivery.
- *Durability:* As for persistence, durability needs depend on the use case. Usually, clients can specify whether they want to issue a durable subscription. This might be relevant for mobile subscribers, e.g., when SPU instances run on mobile devices where links are not stable. With durable subscriptions, events eventually arrive until the connection is intentionally closed.
- *Transactions:* In contrast to traditional messaging scenarios with complex interaction patterns, e.g., presented in [153], many events in event-based applications are self contained and independent of each other. Thus, event bus transaction support does not span across producers and consumers or multiple messages; transactions are rather used to ensures a reliable delivery of single events between brokers and producers/consumers.
- *Order:* Depending on the use case, event-bus-enforced order guarantees might not be necessary. When events contain a time stamp and no strict order guarantees are required, SPU instances can establish an event order or detect out of order events; however, due to time synchronization issues in distributed systems, this is only a best effort approach without strict guarantees.
- *Performance:* Throughput and latency are typically QoS parameters that cannot be configured dynamically; they rather depend on the overall hardware/software setup. The goal is to maximize throughput and minimize latency while guaranteeing, e.g., notification durability. Performance correlates inversely with persistence and reliability; the less persistence and reliability is needed, the higher is the performance.
- *Security*: When notifications contain sensible data, the event bus must ensure that event producers/consumers are authorized to send/receive notifications. At the same time, the event bus itself might not be fully trusted so that notification content must be kept secret while being routed from publishers to subscribers.

Event consumers, like SPUs, should be able to specify different EBS-specific QoS policies for persistence and durability depending on their needs. While persistence addresses broker failures, durability addresses event consumer failures, e.g., due to bad connectivity. We suggest four policies as enhancements to QoS properties in current MOM:

- *Full:* All events require persistence/durability;
- *Recent:* Only the most recent event is relevant and requires persistence/durability;

- *Change:* Only events where values (besides time) change require persistence/durability; and
- *No:* No persistence/durability requirements.

These policies can be specified as additional subscription parameter, i.e., the pub/sub API is extended with QoS properties:

```
1 subscribe(Filter f, QoSpolicy p);
```

An additional QoS property is security. Security is orthogonal to persistence and durability; security demands can apply to each of the above policies. When security mechanisms are supported by the event bus, an SPU has to provide identity information with its subscription. This ensures that only verified subscribers receive notifications. Often, this involves a key exchange procedure and requires multiple steps.

While SPUs rely on the QoS and QoI provided by event producers and the event bus, they are also data and functionality providers themselves. Thus, other system components in the overall event-driven architecture rely on QoS provided by SPUs. QoS aspects of SPUs are related to event stream processing QoS and QoS in SOAs. From QoS properties in event stream processing [42] and in SOAs [25], the following properties are the most relevant in the context of SPUs:

- *Latency:* The time between the arrival of an event at an SPU instance and a resulting action (response time).
- *Throughput:* The number of events per time an SPU instance is able to process.
- *Capacity:* The number of SPU instances that can be active in parallel.
- *Cost:* The computational cost of an SPU instance.
- *Accuracy:* Event stream processing inside of SPUs can be responsible to detect patterns or to evaluate queries on streaming data. Accuracy refers to the number of false positives/negatives in such an event stream processing task.
- *Availability:* SPU instances might be unavailable due to, e.g., connectivity issues. Availability specifies the percentage of time an SPU instance processes events as expected.

As in SOAs, QoS in the context of SPUs result in the specification of Service-level Agreements (SLAs). SLAs describe the QoS properties SPUs provide. Ideally, SLAs are specified in a standardized and machine readable way. This allows to automatically assess the QoS of the overall system. SLAs are especially important when components interact and depend on each other [75]. When one SPU guarantees a high availability but relies on input data from the event bus or another SPU with low availability, it cannot hold its guarantees by design.

SPU instances are containers for application logic; the SPU container model defines automatisms for pub/sub and lifecycle management. The application logic as such can be based upon complex software systems and QoS properties depend also on the libraries used inside of SPUs. Thus, managing SLA interdependencies is a complex topic. Since SPUs are a service-like container model, SLA concepts originating from the service world can be adapted and applied to SPUs.

Monitoring and Benchmarking

In the context of QoS, monitoring and benchmarking are important aspects. Monitoring is performed at runtime while a system is productive. Benchmarking is performed to assess the system behavior under controlled conditions; typically, a standardized workload is used. Based upon benchmarking and the behavior of the system under test, SLAs are defined. During productive use, monitoring mechanisms are used to identify bottlenecks and to check compliance with specified SLAs. Benchmarking and monitoring in SPU environments encompasses the event bus as well as the event stream processing inside SPUs. Lightweight monitoring approaches for (distributed) pub/sub systems exist, e.g., the ASIA system, which performs in-network aggregation of data to minimize the monitoring overhead [77] . Monitoring of SPU instances can be performed by an SPU runtime environment that aggregates data from SPU instances. For this, SPU instances have to implement monitoring capabilities and, e.g., provide information about throughput. The overall system utilization is out of scope of single SPU instances; thus, decisions about distribution of SPUs, e.g., for load balancing purposes, are left to the runtime environment.

Benchmarking of SPU environments involves benchmarking of the pub/sub infrastructure as well as benchmarking the event stream processing in SPUs as such. Benchmarking of pub/sub systems is addressed in existing work [150, 153]. However, most workloads are not designed to address the specifics of event-based systems, e.g., frequent renewal of subscriptions. In [12] we present a workload definition that is tailored to event-based specifics. The focus lies on:

- *Independent Participants:* Event producers and consumers should be logically decoupled. Event producers do not know which SPU instances will receive their events. Thus, events should be self-contained.
- *Dynamic Environment:* Subscriptions should not be static but rather be renewed and changed over time. This reflects SPU instances that are created or shut down over time.

In addition to the pure pub/sub messaging, application logic inside SPU instances is part of the benchmarking process. This requires the workload to define entity-centric event streams and tasks that are applied per entity instance. Real-time shipment tracking and monitoring are such candidates with shipments and vehicles being entities for which SPU instances are created. Since SPUs are event stream processing components, benchmarking can be based upon work on standardized event stream processing benchmarks, which define queries and metrics on event streams [15, 114]. The overall benchmark then integrates a pub/sub workload and an event stream processing workload.

3.4 Network of SPUs

SPU instances consume events, but can also act as event publishers. This enables building SPU networks where SPU instances react on the output of other SPU instances. SPU networks can be used for event enrichment in case entity-centric events are not readily supplied by event producers. An example in the context of a shipment-monitoring scenario is shown in Figure 3.5. The goal is to implement a temperature monitoring of shipments based upon SPUs, i.e., one shipment monitoring SPU instance is created for each shipment. Different shipments can have different temperature thresholds; the shipment-specific threshold is retrieved from a database upon the creation of an SPU instance, i.e., in the `onInstantiation` Method.

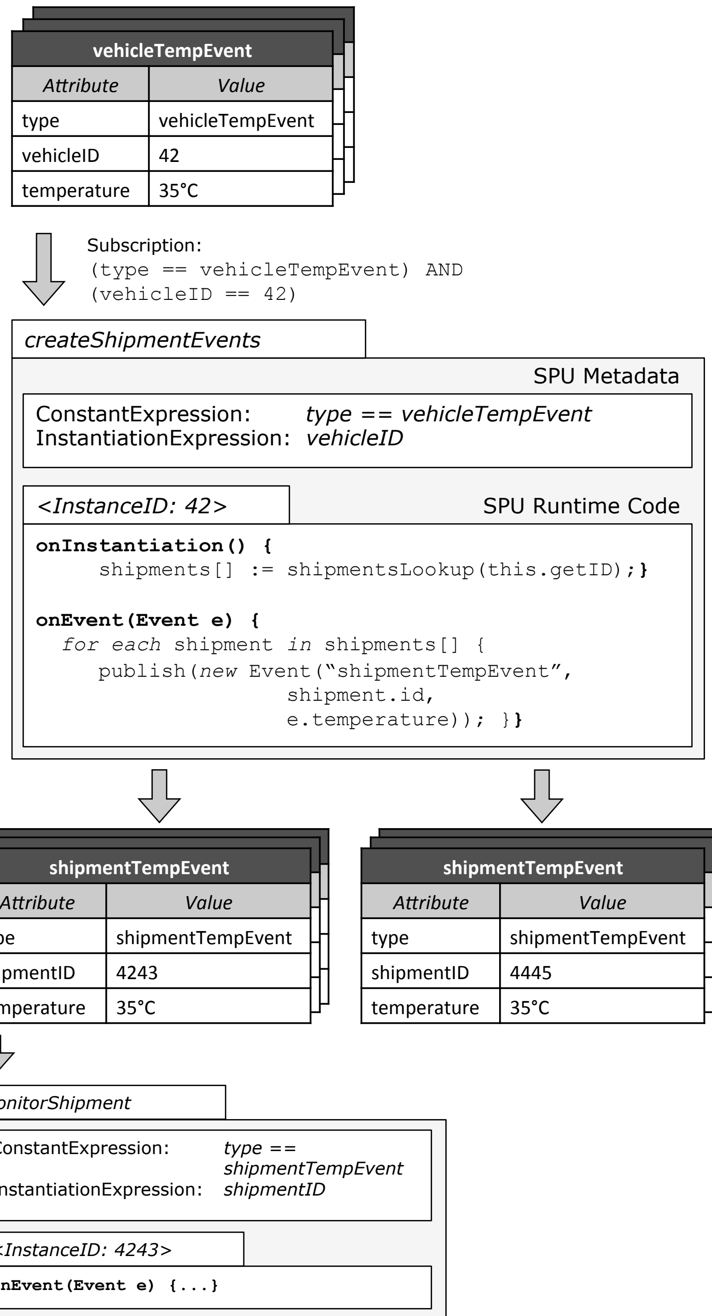


Figure 3.5: Event Stream Processing Units (SPUs) can be used for event enrichment; their output is used as input by further SPUs.

We assume that shipments are transported by vehicles that are equipped with temperature monitoring equipment and a GPS receiver. All fleet vehicles send events that contain the vehicle ID and the current temperature value continuously. These events are not sufficient for a per-shipment monitoring and per-shipment events have to be derived. This can be realized with SPUs: an SPU instance for each vehicle is created that derives and publishes the required per-shipment events. The creation of those `createShipmentEvents` SPU instances can be automated by implicit instantiation strategies, i.e., an SPU instance per vehicle ID is created. The main application logic for deriving per-shipment events is implemented in the `onInstantiation` and `onEvent` methods of the `createShipmentEvents` SPU. The list of shipments that is transported in a vehicle is retrieved at creation of the SPU instance; in Figure 3.5 the SPU instance for Vehicle No. 42 is shown and the list of shipments is retrieved at instantiation. Details of the `shipmentsLookup` method, e.g., the initialization of a database connection, are omitted for brevity. For each `vehicleTempEvent` that arrives at the SPU instance, the `onEvent` method is called. The `onEvent` method publishes one event per shipment that contains the temperature value. The resulting stream of `shipmentTempEvent` events is the source for the `monitorShipment` SPU of which one instance is created per shipment ID.

The advantages of generating derived events with SPUs are the entity-centric abstraction that is applied as well as the inherent distribution capabilities of this approach. Entity-centric abstraction in the presented example means that SPU instances are created for vehicles and for shipments. This follows the OOP paradigm; the direct abstraction of real-world entities fosters an easy understanding of the overall system architecture and functionality. The inherent distribution capabilities of SPUs allow the distribution of SPU instances across different nodes. Since each SPU instance is self-contained and issues its own subscriptions, it can be run on an arbitrary node. Depending on the use case, a high-performance and scalable pub/sub infrastructure is necessary to avoid bottlenecks at the event dissemination layer.

3.5 Instantiation based upon Complex Conditions

The SPU instantiation strategies introduced in Sections 3.2.3 and 3.2.4 refer to the contents of single events only, i.e., stateless evaluation of event content is sufficient. Instantiation is based upon attribute values; an SPU instance is created for each unique set of instantiation attribute values. However, more complex and stateful constant and instantiation expressions are conceptually possible, e.g., event pattern detection. In the following we discuss the applicability of such complex event processing expressions for the creation of SPU instances.

Complex Constant Expression

A non-complex constant expression is applied as filter by all SPU instances; it is a precondition to filter out irrelevant events. Following this semantics, a complex constant expression is treated as a filter and must evaluate to true or false. However, it cannot be evaluated independently per-event anymore. One possibility for the evaluation of complex constant expressions is the creation of SPU instances when the complex constant condition is met, e.g., when a certain event pattern is detected. The instantiation expression can refer to a single attribute or an attribute set as usual. A use case is the creation of SPU instances for shipment monitoring only after a truck has left the logistics hub, i.e., after an event has been seen that notifies about a truck leaving. In this case the constant expression

is a one-time barrier, i.e., once the leaving of a truck has been detected, the condition is fulfilled and SPU instance creation is performed. The constant expression can also be seen as a multi-time barrier, i.e., before each SPU instance creation the condition specified in the constant condition has to be fulfilled.

Complex Instantiation Expression

Instantiation is possible based upon event pattern detection as well as based upon the results of a query defined on the event stream. The instantiation is then based upon the result of the specified query; one SPU instance is created for each distinct result value. However, the challenge is the specification of the SPU instance specific subscription; this subscription has to encompass the complex instantiation expression as well as a filter for a particular result of the instantiation expression that corresponds to the SPU instance. An example is the creation of SPU instances based upon the detection of event patterns. An event pattern detection rule can look as follows:

```
1 Event_A → Event_B → *
```

Event A has to be followed by event B; Event B is followed by an arbitrary event. Matching event sequences are for example:

```
1 Event_A → Event_B → Event_X
2 Event_A → Event_B → Event_Y
3 Event_A → Event_B → Event_Z
```

When an event pattern detection rule is used as instantiation expression, the distinction criterion for the different SPU instances must be identified. In the above example, the last event in the sequence can be used as such an SPU instance identifier. This would result in the creation of three SPU instances that derive their subscription, in this case a pattern detection rule, based upon the instantiation pattern:

```
1 SPU_A-B-X: subscribe("Event_A → Event_B → Event_X");
2 SPU_A-B-Y: subscribe("Event_A → Event_B → Event_Y");
3 SPU_A-B-Z: subscribe("Event_A → Event_B → Event_Z");
```

Which events an SPU instance actually receives when issuing such a subscription depends on the pattern detection engine. The SPU_{A-B-X} might receive events A, B, and X each time a pattern match is detected. It might also receive only the last event of the matching sequence, or just a general notification that the pattern has matched. The different options depend on the applied event consumption policy; preferably, different options should be configurable.

The use of complex constant and/or complex instantiation conditions is not readily compatible with the conceptual idea of our SPUs container model as entity-type-centric encapsulation mechanism for event stream processing application logic. In the above introduced pattern detection example, the generic pattern $A \rightarrow B \rightarrow *$ (complex instantiation expression) corresponds with the notion of an entity type, a concrete match of this pattern, e.g., $A \rightarrow B \rightarrow X$ correspondent with the notion of an entity instance.

The support of complex constant and complex instantiation expressions depends on CEP functionality. For expression evaluation an external CEP engine can be used, or a pub/sub system with support for complex subscription expressions.

3.6 Summary

In this chapter we introduce the SPU container model. SPUs have a simple structure and do not introduce a new language. They provide an encapsulation mechanism for event stream processing application logic on top of a pub/sub system. The developer only has to provide three expressions to identify to which events an SPU applies and what its validity is. This allows implicit/explicit instantiation/completion of SPU instances. Further, developers implement four methods to achieve reactive functionality and SPU lifecycle management. This makes SPUs suitable containers for entity-type-centric event stream processing application logic and provides a clear separation between subscription logic and application logic. Different SPUs can be interconnected to perform distributed event stream processing; we illustrate this in the context of our shipment-monitoring scenario to derive demanded events. The instantiation of SPUs can be based upon complex events, e.g., event patterns. We show how complex constant and complex instantiation expressions can be integrated with our container model; however, they obstruct an intuitive understanding of event stream processing with SPU since the clear relation between SPU instances and entity instances dissolves. We also discuss QoS in the context of SPUs as building blocks in software systems. Different QoS properties determine the processing of event streams inside of SPUs; important is the interface between pub/sub system and SPUs, for which we suggest different QoS policies.

4 Requirements Engineering in EBSs

Event-based components, like Event Stream Processing Units (SPUs), rely on the availability of events. Since event production and consumption is decoupled in Event-based Systems (EBSs), the challenge is to decide upon which events to produce. Research in the event-based field mainly addresses systems consuming events, systems producing events, and systems transporting events. A holistic view that addresses the interdependencies between event production and event consumption from a requirements engineering perspective is an open topic of research. Luckham, for example, assumes events are readily availably [109]; although this is true conceptually, e.g., in news feeds from the Internet, a certain effort is necessary to integrate those events with an event-based application. Chandy recognizes that future potential of events and long deployment cycles have to be taken into consideration [47]; however, no detailed approach is discussed. Thus, we analyze requirements engineering for event-based applications and present an approach to assess event production in the context of enterprise applications.

Requirements engineering is a structured approach to support the construction of IT systems. Goal is the development of a system that fits well with demands. A top-down approach can be applied to achieve this [147]. The requirements engineering process starts with the definition of a goal [171]; from such a goal definition, system specification and design is derived. Scenario analysis on basis of use cases can also be applied for requirements engineering [111]. In general, problem-oriented requirements (goals, use cases) are matched with solution specifications to derive the overall system design [60]. During this process, decision support methods are applied to find the optimal design with respect to, e.g., usability, performance, security, and modularity [145].

At an abstract level, the development of event-based application logic is not different from the development of traditional software components; the implementation of reactive behavior is required to address a goal or a use case. At the technical layer, however, an event-based system architecture exhibits different specifics than pull-based components such as services. In pull-based components, functionality is requested explicitly. Input data is sent to a component as part of a request. Event-based interactions are push-based; functionality is triggered implicitly upon arrival of events. Event production and consumption are decoupled logically; the event producers are not aware of potential event consumers that provide functionality. This fundamental difference influences the requirements engineering process: the production of events has to be addressed explicitly.

In the following we present a top-down requirements engineering principle; we start with identification of top level events required by an event-based application component. For each of those events, we evaluate whether a derivation based upon more fine-grained events is beneficial. We illustrate this approach with a use case study; we then formalize the approach an present a generic requirements engineering methodology.

In EBSs, event consumers express their demands with subscriptions. A pub/sub system delivers matching events that are published by arbitrary producers. The problem that arises is to decide upon which events to detect and to publish. Better sensors, for example, are more expensive but can detect more types of events. These additional event types might be useful in future event-based applications. Since the deployment cycle of sensors can be long, the deployment costs rather than the hardware costs might dominate the overall costs. Thus, the potential for later use of not yet needed event types has to be taken into consideration throughout the requirements engineering process. Better sensors that produce more types of events can be worth the investment when further deployment cycles can be avoided [47].

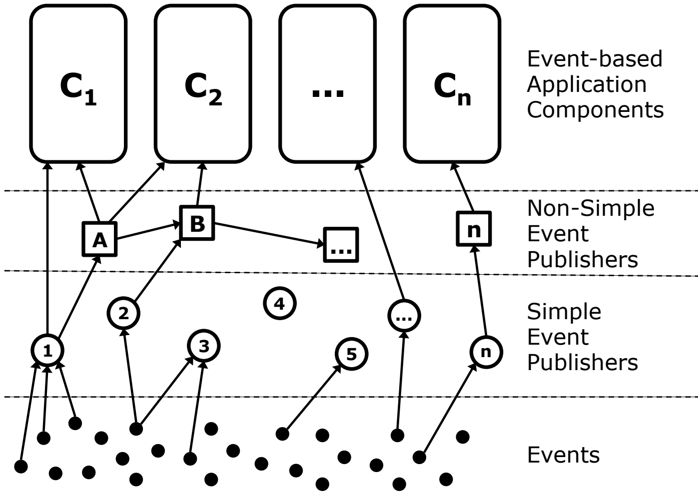


Figure 4.1: Events can be detected directly by simple publishers, or by non-simple publishers that perform event processing.

To address the specifics of event-based interactions in requirements engineering, we introduce an abstract description of event sources and sinks as shown in Figure 4.1. The lowest level depicts the real world. Situations in the real world can be described with events that represent state or incidents. An event that describes state is, for example, a temperature measurement or the current stock price. An event that describes an incident is, for example, the arrival of a truck or a stock acquisition notification. Events can either be physically observable (temperature, truck), or virtual inside systems (stock quotation, stock acquisition). We also distinguish between base events and derivable events. Derivable events are events that can be detected by processing base events. Examples are shown in Table 4.1.

	Physical	Virtual
Base	Temperature; RFID Reading	Stock Quote; Website Access
Derivable	Fire; Shipment Delivery	5% Stock Price Drop; Website Attack

Table 4.1: Event source taxonomy.

We introduce the distinction between physical/virtual base/derivable events to quantify the implications of event detection granularity for the overall requirements engineering process. Figure 4.2 abstractly shows events in a granularity spectrum. The derivable event D_A can be detected by processing base events B_1 and B_2. From another perspective a base event might be derived itself from yet other base events, as it is shown for B_2/D_B. Throughout a requirements engineering process, the granularity at which to detect events has to be specified. The influence of granularity on reuse applies not only to events but is rather general to software engineering [143]. Detecting many fine-grain events will increase the reuse opportunities for single events but introduces additional effort at development, deployment, and runtime. Detecting events too coarse grain limits the potential for reuse opportunities of events; a coarse-grain event is often very specific to a certain use case.

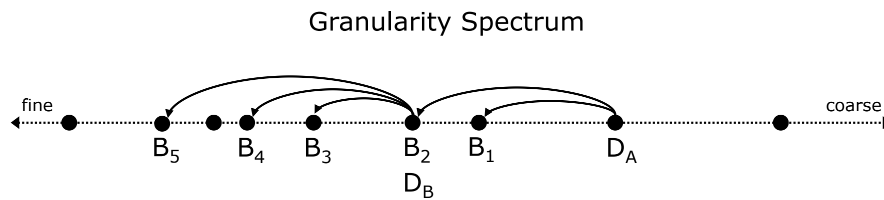


Figure 4.2: Different granularities of derivable (D) and base (B) events: A derivable event can be derived from base events.

At event detection we distinguish between *simple event publishers* and *non-simple event publishers* as shown in Figure 4.1. This distinction allows deriving characteristics that support the requirements engineering process. Real world events can be detected and published by multiple publishers, but they do not have to be detected at all. Further, base and derivable events can be detected by either simple or non-simple publishers. Simple event publishers do not rely on any input events for event detection. Non-simple publishers detect events by means of event processing, e.g., pattern detection; they rely on input from other publishers. Event-based application components can subscribe to events published by either simple or non-simple publishers. Simple event publishers are, for example, sensors; non-simple publishers are, for example, Complex Event Processing (CEP) engines.

Simple and non-simple publishers can be different parts of a single event detection application; the distinction is rather conceptually within the scope of our requirements engineering process. One and the same event can be detected by a simple or by a non-simple event publisher. Detecting the arrival of a train, for example, might be performed by a dedicated sensor that generates this event (simple publisher). It might also be detected by correlating data from multiple sensors. This requires a non-simple publisher that correlates base events; these base events have to be published by simple publishers before. Another example is fire detection: a fire can be detected by a single sensor (simple publisher) that measures temperature, detects smoke, and then publishes an event that indicates the presence of a fire. The reasoning is encapsulated in a black box without the possibility to access the base events (temperature and smoke level). Fire detection can also be performed by a non-simple publisher; it requires temperature and smoke level events produced by simple publishers as input. It then performs the reasoning and publishes a fire event in case of exceeded thresholds for temperature and smoke. When fire detection is performed by a non-simple publisher the required base events might be used in different applications, e.g., to implement a temperature control.

4.2 Requirements Engineering Case Study

To illustrate the process of requirements engineering for event-based applications, we present a case study where we discuss the alternatives for the detection of events. We identify event types required in three related scenarios and discuss implementation possibilities with simple and non-simple event publishers. The overall setting for the scenario is a logistics provider; in this context we analyze an *order-to-delivery process of temperature-sensitive goods* (OD-TEMP), a *fleet management application* (FM), and an *order-to-delivery process with real-time monitoring of environmental conditions* (OD-ENV). These three scenarios require different top-level events:

- Order-to-delivery of Temperature-sensitive Goods (OD-TEMP):
  - Incoming Order Event
  - Shipment Ready Event
  - Shipment Dispatched Event
  - Shipment Delivered Event
  - Temperature Violation Event
- Fleet Management (FM):
  - Capacity Event
  - Truck Mileage Event
- Order-to-delivery with Real-time Monitoring of Environmental Conditions (OD-ENV):
  - Environmental Data Event

4.2.1 Order-to-Delivery Process for Temperature-Sensitive Goods

An order-to-delivery process of temperature-sensitive goods (OD-TEMP) is, for example, required in the supply chain of supermarkets. Supermarkets order products, which are then delivered by a logistics provider. The arrival is acknowledged and completes a process instance. We assume that temperature threshold violations are detected at delivery; potentially spoiled goods are then disposed. The OD-TEMP process is defined based upon different events. These events are either base events or derivable events and can be detected by simple or non-simple event publishers. Event detection is performed, for example, by RFID readers or wireless sensor nodes, but also via user interfaces through which employees report of events.

Incoming Order Event

Supermarkets place orders for goods using an Enterprise Resource Planning (ERP) system. This incoming order is considered a virtual event; it is materialized only inside the system. The event is also considered to be a base event that cannot be derived based upon other events.

- **Simple Publisher:** The event is detected by a simple publisher that interfaces with the ERP system and publishes the event.

Shipment Ready Event

When the ordered goods are prepared and ready for delivery, the shipment ready event is triggered by the person responsible for shipment assembly. The shipment ready event is coarse grain and contains virtual as well as physical information. Physically, the shipment needs to be in place for loading; virtually, the shipment must be registered appropriately with the ERP system to trigger the transport. The event is potentially derivable from base events. It can thus be published by a simple publisher as well as by a non-simple publisher.

- **Simple Publisher:** When treated as simple event, the shipment ready event is triggered by an employee that acknowledges a successful shipment assembly and positioning for loading. From the IT perspective, this requires a user interface, e.g., integrated in the ERP system, to trigger the event upon user interaction.

- **Non-Simple Publisher:** To derive the shipment ready event, different base events have to be combined. First, the assembly of the shipment needs to be tracked, e.g., by RFID scans that produce *RFID scan events*. Once all shipment parts are in place, it must be ensured that the shipment is properly registered with the ERP system and scheduled for pickup (*shipment assembled event*). This check can be automated to some extent; nevertheless, a final check by an employee is desirable (*final check event*). Next, the shipment ready event can be published. The overall effort for deriving the shipment ready event is comparatively high when no base events are readily available. An RFID infrastructure has to be deployed and a user interface to confirm the final shipment assembly is required.

Shipment Dispatched Event

When a shipment leaves the logistics hub, the *shipment dispatched event* indicates this next step in the order-to-delivery process. The shipment dispatched event is associated with the truck leaving the logistics hub and is thus a physical event. It is also derivable and can be published by a simple as well as by a non-simple publisher.

- **Simple Publisher:** The shipment dispatched event can be triggered explicitly when the truck driver confirms that he leaves the logistics hub. For this, the publisher component has to integrate knowledge about shipments loaded in the truck, i.e., it needs to interface with the ERP system to raise the event for each shipment as soon as the driver indicates the departure.

- **Non-Simple Publisher:** The shipment dispatched event can be derived using data about shipments loaded in trucks correlated with truck position changes. This requires trucks equipped with position tracking hardware that publishes *position events*. The logic of event correlation is similar to the simple publisher; the position events have to be correlated with data about shipments in the truck. Information about shipments in the truck can be derived, e.g., from *RFID shipment scan events* that are raised while the truck is loaded.

Shipment Delivered Event

A *shipment delivered event* is triggered when the shipment arrives at the customer. This is a physical base event. It is not reasonable to mark this event derivable since the delivery has to be acknowledged by customers with a signature anyway; this acknowledgment corresponds with the shipment delivered event. Since we categorize the shipment delivered event as base event, the detection is performed by a simple publisher.

- **Simple Publisher:** Typically, customers acknowledge the receipt of a shipment by signing on a mobile device. The simple publisher for the shipment delivered event is integrated with the software on that device. The event is published when the signature is entered.

Temperature Violation Event

When temperature-sensitive goods arrive at the customer it must be guaranteed that a certain temperature threshold was not exceeded. In case the temperature threshold was violated during transport, *temperature violation events* are published to trigger the handling of temperature violation, e.g., disposing the affected goods. Temperature violation events are physical derivable events. Thus, a simple or non-simple publisher can be used for detection.

- **Simple Publisher:** Temperature violation events can be detected using irreversible temperature labels. These labels contain no circuits and indicate temperature violations in an analog way without a timestamp associated; a chemical reaction is triggered at a certain temperature level. When a shipment arrives at a customer the temperature label is checked. A temperature violation is reported by the driver and entered into the system manually.
- **Non-Simple Publisher:** Temperature violation events can be derived automatically when *temperature events* are available. When trucks are equipped with sensors for temperature monitoring, a non-simple publisher takes the temperature monitoring events as input and correlates them with temperature thresholds for shipments. This mechanism also allows temperature violations to be detected faster and to be reported to the driver promptly.

4.2.2 Fleet Management

Vehicles of a logistics provider need to be managed. This involves, e.g., bookkeeping of mileage, capacity utilization, and keeping track of damages. We consider a fleet management application to track mileage and capacity utilization. Such a fleet management application is built upon different events.

Capacity Event

To monitor and optimize the utilization of trucks, capacity utilization of the trucks needs to be tracked. Capacity events inform about the free capacity of a truck when it leaves the logistics hub. Capacity events are physical derivable events that can be published by simple or non-simple publishers.

- **Simple Publisher:** Capacity utilization can be determined and reported manually, e.g., by the driver before a truck leaves the logistics hub. The associated publisher component provides a user interface to enter this data and then publishes the event.

- **Non-Simple Publisher:** The capacity utilization of a truck can also be derived based upon shipment dispatched events correlated with data about the truck. This requires information about shipment sizes and truck capacities. Given this is available, no manual report of free capacity is necessary. In addition, free capacity information is updated with the delivery of shipments. This is an opportunity to reuse free capacity in the short term.

Truck Mileage Event

The mileage is an important indicator to schedule maintenance of vehicles. Mileage events are physical derivable events published by simple or non-simple publishers.

- **Simple Publisher:** The mileage can be reported by the driver. A simple publisher provides a user interface for the driver who reports the mileage at fixed intervals. Another possibility is a direct interface with the truck's on-board unit. The on-board unit keeps track of vehicle data and a simple publisher component can be developed to gather and report this data automatically.

- **Non-Simple Publisher:** The mileage can also be derived by processing *vehicle position events*. By correlating those events the mileage can be calculated automatically and no involvement of an employee is necessary.

4.2.3 Order-to-Delivery Process with Monitoring of Environmental Conditions

As future field of business, the logistics provider might offer the delivery of goods that require auditable tracking of environmental conditions, e.g., fine granular data about temperature and humidity throughout the transport. The overall process is similar to the delivery of temperature-sensitive goods, however with more complex monitoring requirements. Core of this new process are time stamped environmental data events.

Environmental Data Event

To provide auditable environmental monitoring, vehicles must be equipped with appropriate sensors that record environmental data continuously. These environmental data events are physical derivable events detected by either simple or non-simple publishers.

- **Simple Publisher:** Environmental data events contain temperature and humidity data; the data is recorded per truck at fixed intervals. It is sufficient to provide the data offline for analysis after the delivery of shipments. However, it is also possible to make the data available in real-time; this requires a more complex sensor and event publisher setup and generates additional cost.

- **Non-Simple Publisher:** Environmental data events can be derived based upon *temperature events* and *humidity events* that originate from different sensors.

4.2.4 Cost Assessment

Given the described event-based scenarios the next step is to decide upon the realization and implementation details, i.e., to decide upon which events to detect with simple and non-simple publishers. For this, costs are assigned to the events identified in the scenarios; this is shown in Table 4.2. Costs are estimated for the implementation of simple publishers and non-simple publishers. The cost estimates for non-simple publishers assume that base events are available, i.e., the costs for base events have to be summed up for the overall assessment. Table 4.2 also lists the base events required by non-simple publishers.

ID	Event	Simple Cost	Non-Simple Cost	Required Events
1	**Incoming Order**	10	–	–
2	**Shipment Ready**	10	15	3,4,5
3	Shipment Assembled	10	–	–
4	RFID Scan	100	–	–
5	Final Check	10	–	–
6	**Shipment Dispatched**	10	15	4,7
7	Vehicle Position	100	–	–
8	**Shipment Delivered**	10	–	–
9	**Temperature Violation**	25	5	10 or 15
10	Temperature Measurement	100	–	–
11	**Capacity Report**	15	10	6
12	**Truck Mileage**	10	5	7
13	**Environmental Data**	100	5	10,14 or 15
14	Humidity Measurement	100	–	–
15	Temp/Humidity Measurement	110	–	–

Table 4.2: Use case: List of events and dependencies.

To implement scenarios OD-TEMP, FM, and OD-ENV, events 1, 2, 6, 8, 9, 11, 12, and 13 are required (bold in Table 4.2). The decision whether to use simple or non-simple publishers for these events (where possible) is an optimization problem. The costs for different realization strategies can be estimated and compared. In our scenario, the costs for using simple publishers only are 190. When non-simple publishers are used for events 9 and 13, the overall costs sum up to 185 (with event 15 detected by a simple publisher). Thus, deploying combined sensors for temperature and humidity measurement is beneficial in this situation. However, the biggest challenge is to estimate the potential use of events for not yet know scenarios. In our case study, the demand for temperature events in the OD-TEMP as well as in the OD-ENV use case makes the separate detection of temperature and humidity events beneficial; without the OD-ENV use case the cost assessment would not speak for the use of any non-simple publisher. Further, it is important to consider potential future use cases, e.g., providing real-time monitoring data to customers. When sensors for temperature and humidity measurement are deployed that support transmitting of data in real-time, such a service can be realized easily. In case of offline sensors, the demand for real-time data would require the deployment of new sensors.

4.3 Implementation Effort and Reuse Potential of Event Publishers

We now analyze development effort and reuse opportunities for event detection with simple and non-simple publishers from a generic perspective. We present the formalized approach, which we implicitly applied in our use case study. We distinguish between hardware components, e.g., sensor nodes, and software components, e.g., stock tick data providers. We compare simple and non-simple publishers with respect to physical and virtual base and derivable events.

Hardware Development and Deployment Effort

Table 4.3 shows the effort for the deployment and development of event detection hardware. The effort is the same for base and for derivable events given that the detection complexity, e.g., sensor complexity, is comparable. The effort for non-simple publishers depends on already available events. For each base event required by a non-simple publisher, a simple publisher has to be developed and deployed first. When all required base events are already published, the effort is low; when the base event notifications are missing, the respective simple publishers have to be deployed first which results in a high effort.

	Physical Base Event	Physical Derivable Event
Simple Publisher	high effort	high effort
Non-Simple Publisher	*n.a.*	depends on availability of base events

Table 4.3: Hardware development and deployment effort.

Software Development and Deployment Effort

Simple publishers that produce physical events read sensor state and generate the event notifications based upon that data. Simple publishers that produce virtual events need to interface with existing systems that are not necessarily designed to emit events; publishers need to aggregate data that is the foundation for virtual events. Thus, the software development effort when publishing virtual events is higher (see Table 4.4). To detect derivable events (physical or virtual) with non-simple publishers, publishers have to perform event processing to some extent. The development effort of this processing logic is high compared to the detection of base events.

	Physical Base Event	Physical Derivable Event
Simple Publisher	low effort	low effort
Non-Simple Publisher	*n.a.*	high effort

	Virtual Base Event	Virtual Derivable Event
Simple Publisher	medium effort	medium effort
Non-Simple Publisher	*n.a.*	high effort

Table 4.4: Software development and deployment effort.

Hardware Reuse Potential

The potential for hardware reuse is low for derivable events that are detected by simple publishers (see Table 4.5). Dedicated hardware needs to be deployed to detect events that could also be detected by applying event processing techniques to base events. When derivable events are detected by non-simple publishers, fine-grain base events are used. This increases the reuse potential; some or all base events may be used to derive different high-level events.

	Physical Base Event	Physical Derivable Event
Simple Publisher	high potential	low potential
Non-Simple Publisher	*n.a.*	high potential

Table 4.5: Hardware reuse potential.

Software Reuse Potential

The low reuse potential of hardware in case of derivable events detected by simple publishers carries over to the reuse potential of software (see Table 4.6). The application logic to detect derivable events with simple publishers is designed for a certain purpose and there is a high chance that it needs to be adapted to fit the demands of other applications. The reuse potential of the simple publisher detection logic for virtual events is higher than for simple publishers of physical events; the code is not tightly coupled with a certain hardware, e.g., a wireless sensor node, and best practices in software engineering enable code reuse. The reuse potential of non-simple publishers that detect derivable events is higher compared to the reuse potential of simple publishers. Some parts of the code can be generic, e.g., functions for event correlation, and can be reused when developing other non-simple publishers. This is also the main concept of CEP engines: rules to derive events are specified in a query language rather than implemented manually.

	Physical Base Event	Physical Derivable Event
Simple Publisher	medium potential	low potential
Non-Simple Publisher	*n.a.*	medium potential

	Virtual Base Event	Virtual Derivable Event
Simple Publisher	high potential	low potential
Non-Simple Publisher	*n.a.*	high potential

Table 4.6: Software reuse potential.

4.4 Requirements Engineering Process

Based upon the qualitative considerations presented in Section 4.3 we present a quantitative approach to trade off event detection with simple publishers against event detection with non-simple publishers. This decision depends on the costs for event detection hardware and software, on the deployment effort, as well as on the reuse potential of events. However, especially the reuse potential is difficult to quantify without knowledge of future demands. An advantage of the event-based

approach – the decoupling between event producers and consumers – becomes a challenge in the context of requirements engineering.

The reuse potential of events increases the more fine granular they are. Thus, the requirements engineering process should assess different possibilities for the detection of events, i.e., for derivable events the detection with simple and non-simple publishers should be considered. Figure 4.3 shows a cost estimation process for event-based applications. First, the events required at the application level are identified. It is then checked for each event whether it can be derived from other events; the required base events are then determined. As shown in Figure 4.2, these events can be derivable as well and the process step is executed repeatedly. During this recursive event identification process, the dependencies between derivable events and base events are tracked, i.e., for each derivable event the required base events are recorded. The identification of base events should be stopped when the detection overhead obviously exceeds the reuse potential, e.g., when hardware for event detection is far too costly to be deployed at large scale. The next step in our requirements engineering approach is the estimation of costs for the detection of events. This step is twofold for derivable events: derivable events can be detected by a simple or non-simple publisher and costs can be estimated for either option. Cost estimations are based upon the hardware acquisition and deployment costs as well as on the software development and deployment costs. It has to be taken into account that costs consist of a static part that occurs once (initial development and deployment costs) and a dynamic part with reoccurring expenses (maintenance costs). The cost estimates for non-simple publishers do not contain the costs for the required base events but only costs for event processing required for the event derivation process. Costs of base events are taken into consideration during the final cost assessment.

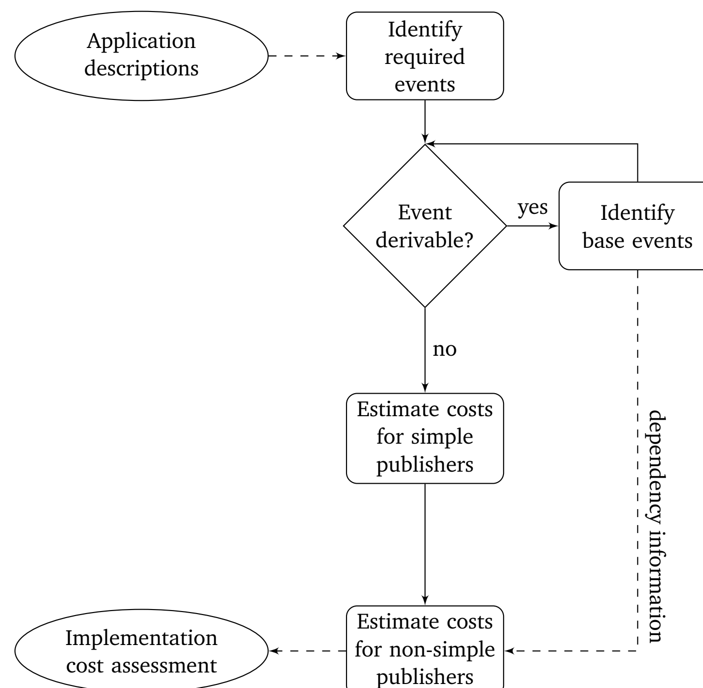


Figure 4.3: Requirements engineering process to assess costs of implementation alternatives.

The estimation process results in a list of event types along with properties for each event as shown in Table 4.2:

- Event ID to reference each event;
- An indicator to mark events as required at the application layer;
- Cost estimate for detection with simple publisher;
- Cost estimate for detection with non-simple publisher (if applicable);
- Events required as input for non-simple publisher.

Based upon the estimated costs, the use of simple or non-simple publishers can be compared for each event type. When a non-simple publisher is used, the costs for the detection of the required base events have to be added to the overall costs. When base events are reused, these costs only count once. During the cost assessment process the costs for different scenarios can be compared. As a result, a decision for each event type is made whether to detect it with a simple or a non-simple publisher. This also includes a list of base events that needs to be detected as source for non-simple publishers. It should be noted that multiple alternative implementations (with different costs) for non-simple publishers are possible, i.e., non-simple publishers that publish the same derived events but require different events as input.

The process of assessing implementation alternatives for event detection based upon cost estimates is only part of the requirements engineering process. Further, functional and non-functional properties that are not necessarily included in the estimated costs should be considered as well:

- While the reuse potential for fine-grained events is high, the overhead for event processing increases. Multiple fine-grained events have to be published to allow the derivation of a high-level event. This increases the network traffic, the latency, and the demand for computational resources. This has to be taken into account in the requirements engineering process. When, for example, high performance is demanded by an application, the costs for simple and non-simple publishers should be adapted accordingly.
- An important factor in event-based applications is the simple reuse of events; an application component interested in events simply issues a subscription; different consumers do not interfere. However, this easy access to events is also challenging: when developing and deploying event publishers, the potential for future use should be estimated as well. Especially in the context of event detection hardware, e.g., sensors, the deployment costs can be significantly higher than hardware costs. It might be advisable to invest in more powerful hardware but saving future deployment costs. Thus, visions should be developed for future event-based applications so that reuse potential for events as well as the upcoming need for certain event types can be taken into consideration.

4.5 Summary

The requirements engineering process presented in this chapter supports decisions regarding the granularity of event detection in event-based applications. The process starts with the identification of required top-level events; it enables the assessment of implementation alternatives for event

publishers. Simple or non-simple event publishers can be used for event detection. This results in different costs and reuse potential. The more fine granular event detection is, the higher are the chances for reuse; however, the higher are also the costs. Further, it is important to incorporate the potential for future use of events.

Various mechanisms exist to guide software engineering decisions [145]. Our process is a basic approach without a detailed cost estimation guideline. However, the basic tradeoffs have been illustrated; they are independent of a concrete cost model. Our approach proceeds top-down, starting from top-level events. In addition, it can be beneficial to look at event detection bottom up to identify future use cases: based upon potentially available events, use cases can be derived. Thus, identifying detectable events bottom up might influence the top-down requirements engineering process.

5 SPU Integration with Business Processes

The Internet of Things and Cyber-physical Systems provide enormous amounts of real-time data in form of streams of events. Businesses can benefit from the integration of this real-world data; new services can be provided to customers, or existing business processes can be improved. Events are a well-known concept in business processes. However, there is no appropriate abstraction mechanism to encapsulate event stream processing in units that represent business functions in a coherent manner across the process modeling, process execution, and IT infrastructure layer.

In this chapter we apply the Event Stream Processing Unit (SPU) model to integrate event stream processing with business process modeling and execution. The encapsulation of event stream processing logic in SPUs enables a seamless transition between process models, executable process representations, and components at the IT layer. SPUs are a generic concept not limited to a specific modeling notation and we introduce Business Process Model and Notation (BPMN) and Event-driven Process Chain (EPC) extensions to model SPUs. BPMN is the newer modeling notations – BPMN 1.0 has been introduced in 2004 [38], BPMN 2.0 in 2011 [130] – and supports modeling from the abstract business perspective as well as from the technical perspective. EPCs – introduced in 1992 [96] – focus on abstract process models from the business perspective; they are common in industry due to their earlier availability. EPC models are transformed into a technical process representation for execution, for example in a BPMN model. Thus, our BPMN extensions are also the foundation for EPC model execution and we present a mapping between SPUs in EPCs and SPUs in BPMN.

In the following we derive requirements for the integration of event streams with business processes. We investigate the applicability of BPMN and EPCs for modeling event stream processing. We introduce Event Stream Processing Tasks (ESPTs) as BPMN 2.0 extension to model SPUs at the technical process layer. We introduce Event Stream Processing Services (ESPSs) as an extension to EPCs to model SPUs at the abstract business layer. Our integration of SPUs with BPMN and EPCs includes a transformation approach: we present a mapping of SPU-containing process models from EPCs to BPMN and from BPMN to the Business Process Execution Language (BPEL). We also show how SPUs in process models are linked to the IT infrastructure layer. We present the implementation of our EPC and BPMN extensions in Software AG ARIS and we sketch the model-to-execute workflow that brings SPU-containing business process models to execution. We illustrate our approach using the previously introduced logistics process as running example.

Scenario

The processing of an order consists of multiple process steps: an order is received, the invoice for the order is prepared and the payment is processed. With SPUs, data generated during the physical transport can now be integrated with this process. An event stream that provides monitoring data related to the shipment can be used to detect, e.g., temperature threshold violations. An SPU can represent such a monitoring task and integrate it at the business process modeling, business process execution, and IT infrastructure layer. A shipment monitoring SPU is *instantiated* with the shipment

of an order. The SPU *completes* after delivery. Throughout this section, we illustrate our approach on the basis of such a monitoring SPU.

5.1 Event Stream Integration Requirements

While single events are a well known and established concept in business processes [130,170], event stream processing lacks an appropriate abstraction for the seamless integration across the process modeling, process execution, and IT infrastructure layer. In collaboration with Software AG we applied SPUs as such an integration concept for event stream processing. SPUs provide a service-like abstraction to encapsulate event stream processing logic at the abstraction level of business functions. They hide implementation details and are suitable building blocks for the integration of event stream processing with business processes. With the encapsulation of event stream processing in SPUs a high coherence between the different layers is achieved; as discussed in Section 2.2 this supports a seamless transition between model, executable workflow, and IT infrastructure [3,134].

We analyze business process modeling, business process execution, and the IT infrastructure, and derive requirements for SPUs at the modeling, execution, and IT infrastructure layer. We address the decoupled nature of event-based systems and provide process modelers with an appropriate representation of SPUs that can be mapped to executable workflow representations and the IT infrastructure seamlessly. SPUs encapsulate event stream processing logic at the abstraction level of business functions and hide implementation details. At the IT layer, SPUs are manageable components that are conceptually equivalent to services in a SOA. SPUs contain, for example, Complex Event Processing (CEP) functionality.

5.1.1 Business Process Modeling Layer

Process models are typically created by business experts that have a good knowledge about the company structure and established workflows. These process models describe interactions between business functions [137]. For a clear separation of concerns between the business perspective and the IT perspective, it is necessary to encapsulate event stream processing logic. This encapsulation is provided with SPUs that hide technical details at the modeling layer. SPUs are the abstract representation of business functions that process event streams. SPUs require at least one event stream as input and may output event streams or single events. An important characteristic of SPUs is the demand for continuous processing of event streams; rather than in single request/reply interactions, SPUs process new events as they arrive, e.g., a shipment monitoring SPU receives new monitoring data continuously.

Requirements

For the integration of event streams, the modeling notation has to provide *elements or patterns to express SPUs (R_1)* [48]. While the actual event-processing functionality is encapsulated inside SPUs, event streams – as they represent a core data asset – should be accessible by the modeler [116]. Further, integrating event streams during modeling simplifies the transition to an executable workflow [21]. Thus, the modeling notation has to provide *means to express event streams as input/output to/from SPUs (R_2)*. Finally, the modeling notation must allow *SPUs to run continuously and in parallel*

to other tasks (R_3). This includes *appropriate execution semantics adapted to event-based characteristics (R_4)* [64]. These requirements are structured in Table 5.1.

R_1	Model notation support for SPUs
R_2	Model notation support for event streams
R_3	Continuous and parallel execution of SPUs
R_4	Event-based-compliant SPU completion semantics

Table 5.1: Requirements at business process modeling layer.

5.1.2 Workflow Execution Layer

The execution of business process models requires a transition from the, often graphical, model notation to a formal process representation. The interactions between the different process tasks are formalized in a workflow description, e.g., using BPEL. This workflow description contains, e.g., service invocations and defines the input data for services. Like traditional business process tasks can be mapped to human tasks or service invocations, ESPTs need to be mapped from the model to the IT infrastructure. Important is the definition of input data, so that the workflow is executable automatically.

Requirements

To support SPUs at the workflow execution layer, the execution notation has to *support the instantiation of the SPU containers provided by the IT infrastructure (R_5)* [9]. It further needs means to *define streams of events as input and output of SPUs (R_6)*. The *instantiation and completion of SPUs needs to be configurable with respect to event-based characteristics (R_7)* [64]. These requirements are structured in Table 5.2.

R_5	SPU instantiation support
R_6	Support input/output to/from SPUs in form of event streams
R_7	Control over SPU instantiation and completion behavior

Table 5.2: Requirements at workflow execution layer.

5.1.3 IT Infrastructure Layer

The IT infrastructure holds the technical representations of SPUs. It is responsible for the execution of the encapsulated event stream processing logic when demanded throughout a workflow. In contrast to SOA services, SPUs follow the event-based paradigm. Services are invoked explicitly, while SPUs react to events. Services encapsulate business functions in a pull manner (reply is requested) while SPUs encapsulate reactive business functions that are defined on events pushed into the system.

Requirements

The IT infrastructure has to *provide a runtime environment for SPUs that respects event-based characteristics, e.g., implicit instantiation (R_8)* [64]. It must *provide containers for SPUs that represent business functions (R_9)* [48]. Just like services, these *SPU containers must be manageable and capable of receiving the required data in form of event streams (R_{10})* [136]. The requirements are structured in Table 5.3.

R_8	Runtime environment for SPUs
R_9	SPU container model
R_{10}	Support for SPU management

Table 5.3: Requirements at IT infrastructure layer.

5.2 Event Stream Processing Units in Business Processes

In this section, we present our approach for a seamless integration of event streams with business processes. Our solution addresses the identified requirements and spans the three business process implementation layers. It provides a coherent encapsulation of event stream processing logic in components that constitute business functions. To support SPUs at the business process modeling, business process execution, and IT infrastructure layer, we suggest mechanisms at each layer.

At the modeling layer, we introduce *Event Stream Processing Tasks* (ESPTs) to represent SPUs in BPMN process models; we introduce *Event Stream Processing Services* (ESPSs) to represent SPUs in EPCs. At the IT infrastructure layer, Eventlets are the implementation of SPUs. The execution layer is responsible for the mapping between ESPTs and Eventlets. This is shown in Figure 5.1: like services are the basic building blocks of a SOA, SPUs are the basic building blocks of an Event-driven Architecture (EDA). At the execution layer, service tasks in a model are mapped to, e.g., web services. Equally, ESPTs are mapped to Eventlets.

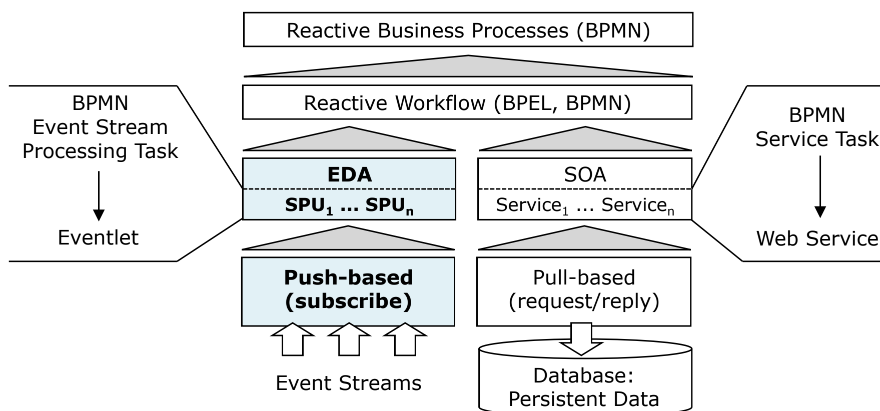


Figure 5.1: The execution of SPU-containing BPMN process models requires a mapping from the modeling layer to technical components on the pull- and push-side.

We apply an additional transformation step for the execution of EPC models. We introduce a mapping of ESPSs in EPCs to ESPTs in BPMN; this is shown in Figure 5.2. The nomenclature of an *EPC*

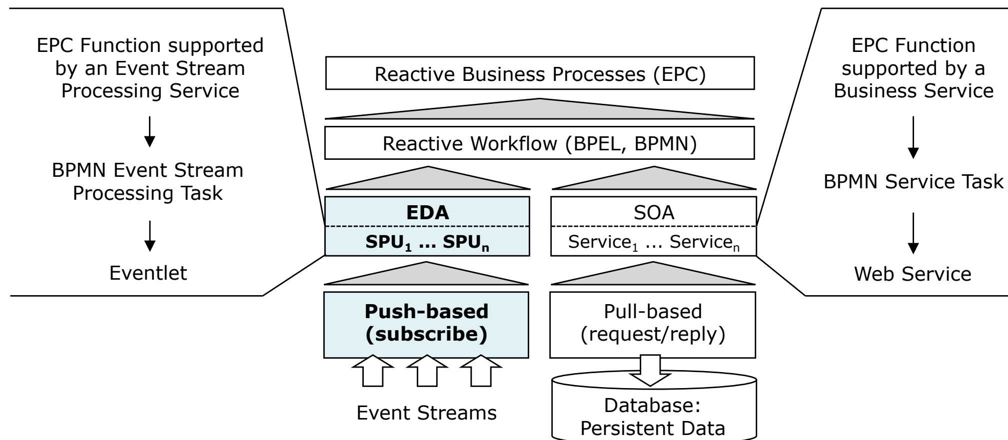


Figure 5.2: The execution of SPU-containing EPC process models requires an additional transition from the abstract EPC model to a technical BPMN model.

functions supported by an ESPSs is EPC-specific; it describes the function type as we will detail in the following sections.

5.2.1 Modeling Layer

Different business perspectives need to be incorporated during business process modeling. Business analysts provide knowledge about processes from an abstract business perspective. The resulting process models are then refined iteratively, e.g., by process engineers, and evolved to more technical process representations. EPCs, for example, follow a strict scheme with business events followed by functions. They are widely adopted in industry and well suited for the abstract modeling from the business perspective [115]. BPMN is a newer and more powerful process modeling notation. It supports abstract as well as technical process models and is typically more compact than EPCs. Both notations can be combined to support a holistic BP modeling process. This is, for example, the case in Software AG's model-to-execute process: EPCs are used for abstract, business-oriented models; BPMN is used for technical process models. A model transformation process is specified for the transition between EPCs and BPMN.

The integration of SPUs with BPMN and EPCs requires extensions to the modeling notations that address the characteristics derived from the streaming nature of event data. SPUs exhibit the following specific properties that cannot be expressed completely with existing BPMN and EPC elements:

- *Execution semantics:* After the instantiation, SPUs can run indefinitely; events arrive and are processed continuously, e.g., temperature measurements during the shipment transport. The completion semantics differ from service-like request/reply interactions where the reply triggers the process control flow to proceed. In contrast, completion of SPUs has to be triggered - either implicitly or explicitly. In either case, the completion indicates a clean shutdown. *Implicit completion* requires the specification of a condition that determines when the SPU should complete; this condition is evaluated internally by SPU instances. Examples are a timeout in case no new events arrive, the detection of a certain event pattern, or a dedicated event, e.g., a shipment delivered event. *Explicit completion* triggers the completion of an SPU externally. When

a process reaches a point where the processing of an event stream is not required anymore the shutdown of an SPU is triggered, e.g., when the shipment arrival has been confirmed.

- *Signaling:* The continuous processing inside of SPUs requires support to trigger concurrent actions, e.g., triggering exception handling in case of a temperature threshold violation without stopping the shipment monitoring SPU.
- *Event stream input and output:* The inputs for SPUs are event streams. An event stream is specified by a subscription to future events, e.g., temperature measurements for a certain shipment. The output is specified by an advertisement that describes the events producible by an SPU.

SPU Integration with BPMN

BPMN is widely adopted in industry and has a broad tool support. From a technological perspective, processes can be modeled in different granularities with BPMN. From a semantical perspective, the single building blocks (BPMN tasks) of a process model should reflect business functions and hide technical details. Our extensions to BPMN are shown in Figure 5.3. We define *Event Stream Specifications* (ESSs) that reflect input data and output data in form of event streams. Further, we introduce *Event Stream Processing Tasks (ESPTs)* to model SPUs.

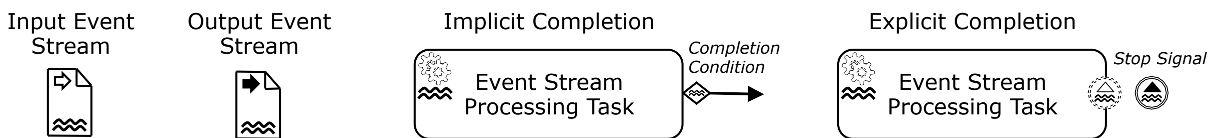


Figure 5.3: Extensions to BPMN: Event Stream Specifications (ESSs) and Event Stream Processing Tasks (ESPTs).

Definition: Event Stream Specification

*In BPMN an **Event Stream Specification (ESS)** (→ R_2) references a stream of events and their parameters. ESSs can be used as input and output of ESPTs. An ESS used as input determines the subscription an ESPT has to issue. An ESS used as output determines the advertisement that describes the event output stream of an ESPT.*

Definition: Event Stream Processing Task

*In BPMN an SPU is modeled as **Event Stream Processing Task (ESPT)** (→ R_1,R_3,R_4). An ESPT requires at least one ESS as input. It may have output ESSs. When the control flow reaches an ESPT, it is activated with the specified ESS as input. The transition from the active state to the completing state (see BPMN task lifecycle [130, p. 428]) is triggered implicitly or explicitly (→ R_5).*

The implicit completion of an ESPT is realized with a modified conditional sequence flow; the condition determines when the ESPT completes. The explicit completion is realized with a dedicated signal. It is attached as non-interrupting signal to the boundary of the ESPT. Upon completion, either implicitly or explicitly, the ESPT stops processing, performs a clean shutdown, and passes on

the control flow, i.e., no additional token is created at catching the shutdown signal. To trigger concurrent actions, ESPTs can activate outgoing sequence flow elements while remaining in the active state. Non-interrupting conditional events can be attached to the boundary of an ESPT to indicate a conditional activation of sequence flow elements. An ESPT can be modeled with combined explicit and implicit completion.

Related BPMN Concepts

Events are part of the BPMN specification. However, events in BPMN are meant to affect the control flow in a process [130, p. 233]. Events modeled as ESS do not exhibit this property; they are rather a source of business relevant information that is exploited within the process. Due to these different semantics, events in the sense of the BPMN standard are not appropriate to model SPUs.

From the task types contained in the BPMN standard, service tasks, business rule tasks, loop service tasks, and multiple instance service tasks share properties with SPUs. However, the modeling of SPUs with those existing BPMN task types is not feasible:

- *Service Tasks* are containers for business functions, that are implemented as SOA services. The execution semantics for service tasks state, that data input is assigned to the service task upon invocation; upon completion output data is available. For SPUs, this separation is not feasible; input data is arriving continuously and output data can be available during task execution in form of output streams. Therefore, service tasks are no appropriate representation for SPUs.
- In *Business Rule Tasks*, event stream processing can be used to check conformance with business rules. However, event stream processing supports a wider application spectrum than conformance checking, e.g., real-time shipment tracking. Further, output in form of event streams is not part of business rule tasks; their purpose is to signal business rule evaluation results.
- *Loop Service Tasks* perform operations until a certain stop condition is met. However, the whole loop task is executed repeatedly, i.e., a repeated service call. This repeated execution of a business function depicts a different level of abstraction compared to continuous processing inside an SPU; SPUs perform continuous processing to complete a single business function. To use loop tasks for event stream processing, the process model would have to define the handling of single events rather than the handling of event streams. This conflicts with the abstraction paradigm of business functions and degrades coherence across the layers.
- *Multiple Instance Service Tasks* allow the execution of a task in parallel, i.e., parallel service calls. However, like loop tasks, this would require one task per event which conflicts with the intention to encapsulate business functions in tasks. In addition, the number of task instances executed in parallel is static and determined at the beginning of the task. This is not suitable for event processing since the number of events is not known a priori.

In general, BPMN tasks have no support for triggered completion required in event processing. In addition, event streams cannot be represented as input to and output from tasks. Thus, we extend BPMN with ESPTs. ESPTs support implicit and explicit completion, an essential part of SPU execution semantics. Further, we introduce ESSs as input to and output from ESPTs in the form of event streams.

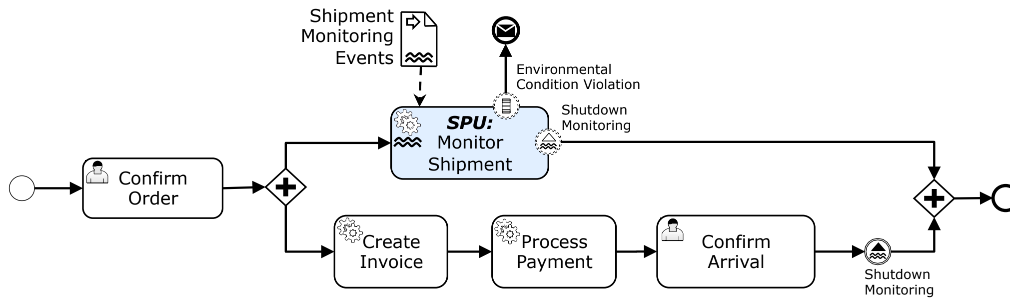


Figure 5.4: Shipment monitoring SPU that is stopped explicitly. The data input/output of the service tasks omitted.

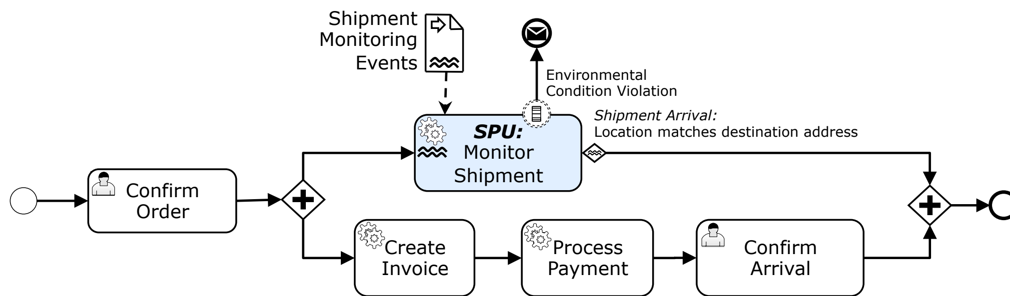


Figure 5.5: Shipment monitoring SPU that is stopped implicitly. The data input/output of the service tasks omitted.

Example: Modeling Shipment Monitoring with BPMN

To illustrate the application of our BPMN extensions, we model the monitoring of environmental conditions in the order process introduced at the beginning of Chapter 5. Figures 5.4 and 5.5 show two variants with different completion strategies. The shipment monitoring is an SPU that receives monitoring events as input stream. This *shipment monitoring SPU* is modeled as an ESPT in BPMN; the monitoring events are assigned as an input ESS. Attached at the boundary of the monitoring task is a conditional event. The outgoing control flow indicates a violation of environmental conditions, e.g., temperature threshold exceeded. The message event at the end of this control flow path can activate a task or trigger a different process for handling the exception.

In Figure 5.4, the shipment monitoring is modeled with explicit completion semantics. As soon as the shipment has arrived, the monitoring is not required anymore. Thus, the monitoring task completion is triggered by sending the stop signal. In Figure 5.5, the shipment monitoring is modeled with implicit completion semantics. This requires the definition of a completion condition. In our example, we specify the shipment arrival: when the location of the shipment matches the destination address, the monitoring is completed. Other implicit completion conditions could be dedicated arrival events, e.g., arrival scans of shipment barcodes, or timeouts, e.g., no new monitoring events for the shipment arrive. The condition needs to be evaluated inside the SPU, thus support for different condition types depends on the technical infrastructure that executes SPUs. The modeling of data flow between ESPTs is shown in Figure 5.6. The shipment monitoring SPU outputs a stream of events that indicate exceeded temperature thresholds. The *report threshold violation* SPU receives these events and implements the reporting, e.g., by sending an email in case temperature thresh-

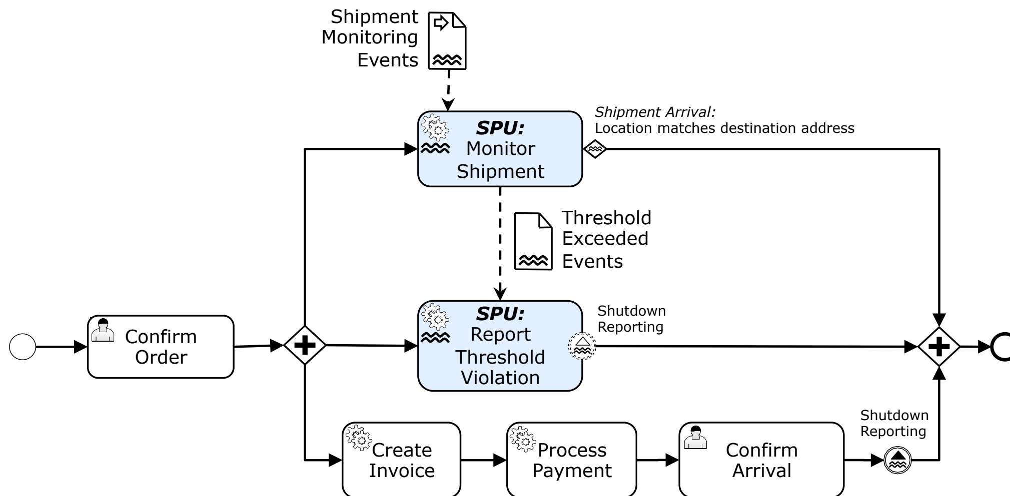


Figure 5.6: Interacting SPUs: Output from monitoring SPU is used as input for reporting SPU. The SPUs have different completion strategies.

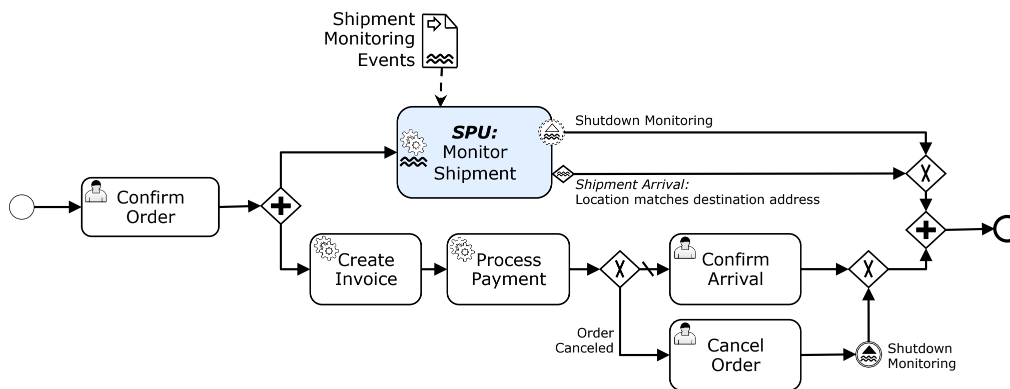


Figure 5.7: ESPT with implicit and explicit completion: Upon cancelation of an order the shipment monitoring is completed explicitly; upon shipment arrival, monitoring completes implicitly.

olds were exceeded multiple times. Completion of the reporting SPU is triggered explicitly after the confirmation of arrival of the shipment. The monitoring SPU completes implicitly.

ESPTs can also be modeled with implicit and explicit completion in parallel as shown in Figure 5.7. The implicit completion is the default case: the monitoring stops as soon as the shipment has reached its destination. In addition, an explicit completion is modeled: when a customer cancels the order, shipment monitoring becomes obsolete. In this simplified version of the process, the monitoring would end anyway since the process ends after the cancelation; however, in more complex scenarios more process steps follow the cancelation. The output of ESPTs can also affect process execution as shown in Figure 5.8. When an environmental condition violation is detected the shipment monitoring is stopped. After completion of the shipment monitoring ESPT, discarding of goods is triggered and the customer decides upon cancelation of the order. In this case a reimbursement is triggered via a compensation event.

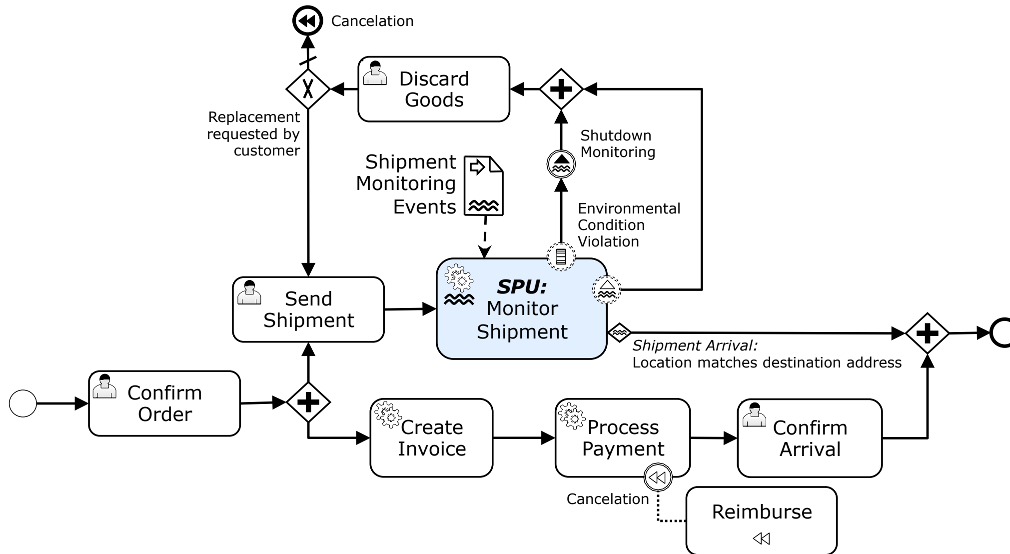


Figure 5.8: ESPT that affects process execution: Upon detection of an environmental condition violation the shipment monitoring is stopped and the customer decides upon cancelation of the order.

SPU Integration with EPCs

EPCs are a notation for Computation Independent Models (CIMs), a concept in model driven architectures [118]. CIMs, also referred to as business or domain models, are created from a business viewpoint and contain only few technical details. EPCs became popular as process modeling notation for SAP R/3 as well as notation in the context of ARIS (Architecture of Integrated Information Systems) [155]. ARIS is an approach for holistic business process modeling and management of enterprise information systems. EPCs consist of functions (e.g., confirm order) preceded and followed by business events (e.g., order arrived and order confirmed). Event stream processing can be modeled as such EPC functions; an EPC function refers to an SPU. In ARIS, EPC functions are supported by services with attached capabilities, e.g., an order confirmation service has the capability to send out confirmations [162]. To model SPUs in EPCs we specify an appropriate supporting service type for EPC functions that represents event stream processing. This involves extensions to EPCs on the basis of service type objects with capabilities.

Details of a service type object are modeled in a service allocation diagram; it describes a service from an abstract point of view. In the service allocation diagram arbitrary objects are connected to the service type object via associations. Connected objects are, for example, descriptions of the organization (organizational unit, responsible person) or data objects used as input or as output of the service.

We introduce *Event Stream Processing Services* (ESPSs) to support EPC functions that represent event stream processing. ESPSs are a distinct service category based upon service type objects [162]; they are represented with a custom symbol type (see Figure 5.9, center). We introduce *Event Stream Specifications* (ESSs) that reflect input data and output data in form of event streams. We adapt

a cluster model object to represent these event streams (see Figure 5.9, left). The Event Stream Processing Unit type is used to refer to the technical realization of SPUs (see Figure 5.9, right).

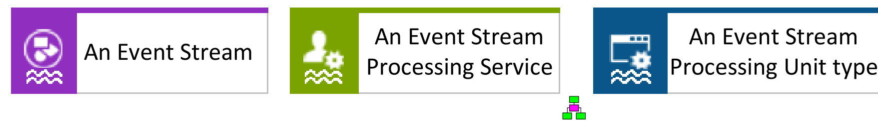


Figure 5.9: Extensions to EPCs: Event Stream Specification (ESS), Event Stream Processing Service (ESPS), and Event Stream Processing Unit Type.

Definition: Event Stream Specification

*In EPCs an **Event Stream Specification (ESS)** ($\rightarrow R_2$) references a stream of events. An ESS is a subtype of an abstract business object. The object type can be used as input or as output of functions, ESPSs, or Event Stream Processing Unit types. The attached connection type specifies whether the ESS is input to or output from other objects. An ESS used as input determines the subscription an ESPS has to issue. An ESS used as output determines the advertisement that describes the event output stream of an ESPS.*

Definition: Event Stream Processing Service

*A function in an EPC that refers to an SPU is supported by an **Event Stream Processing Service (ESPS)** ($\rightarrow R_1, R_3, R_4$). An ESPS requires at least one ESS as input. It may have output ESSs. When the control flow reaches a function supported by an ESPS, this ESPS is activated with the specified ESS as input. The completion of the ESPS is triggered implicitly or explicitly ($\rightarrow R_5$).*

Implicit and explicit completion of ESPSs is expressed with different instantiation capabilities: Start Processing with Completion Condition and Start Processing. Explicit completion is also expressed as a distinct capability of the ESPS. Upon completion, either implicitly or explicitly, the ESPS stops processing, performs a clean shutdown, and passes on the control flow. To trigger concurrent actions, ESPSs can send events; this is modeled as a loop. An ESPS can be modeled with combined explicit and implicit completion. An ESPS has an associated Event Stream Processing Unit type: it provides the link to technical process model representations. The Service Allocation Diagram for an ESPS is shown in Figure 5.10.

Related EPC Concepts

Events are integral notation elements in EPCs. In contrast to BPMN, where events are optional, EPCs require a strict sequence of events followed by functions. Events in EPCs, however, depict process state and define the process control flow. As in our BPMN model extensions, event streams represented by Event Stream Specifications (ESSs) have different semantics; they represent data input/output to/from functions. Thus, regular EPC events are not a suitable representation for event streams as data source for event stream processing.

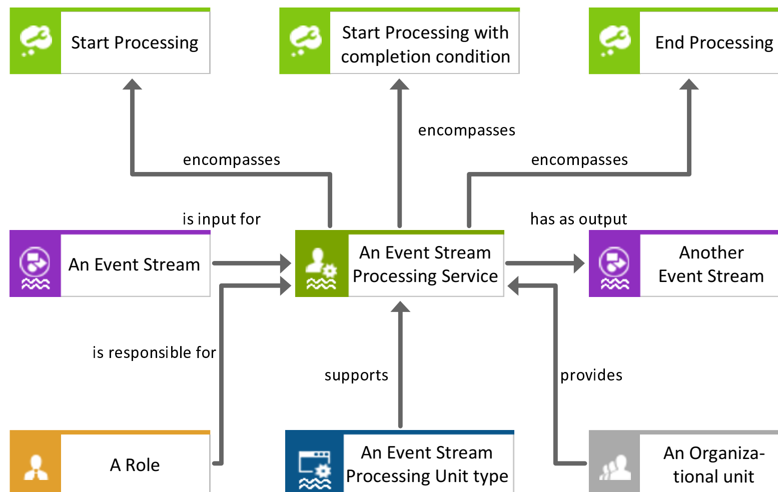


Figure 5.10: A Service Allocation diagram provides the abstract configuration of an Event Stream Processing Service.

Business activities are represented by EPC functions. The EPC specification does not contain different function types comparable to BPMN task types. Rather, EPC functions are supported by service types, which detail the execution of functions. These function-supporting service types have to be specified explicitly. Modeling tools, like Software AG ARIS, contain a repository of service types; a service type that represents event stream processing, however, is not provided and thus specified above. Since we rely on BPMN for the execution of EPCs, our event stream processing service type encompasses ESPT execution semantics.

In EPCs functions can be triggered via events. But in contrast to BPMN, where events can be attached to the border of activities, events in EPCs must be placed in front of or after functions. Thus, we use a loop to model the sending of events; such events can then trigger further functions.

Example: Modeling Shipment Monitoring with EPCs

To illustrate the application of our EPC extensions, we model the monitoring of environmental conditions in the order process introduced in the beginning of this chapter with EPCs. An SPU is used to process an event stream that contains environmental data events, e.g., temperature measurements. Figure 5.11 shows the service allocation diagram for the ESPS that represents this shipment monitoring SPU. The shipment monitoring ESPS requires shipment-monitoring events as input; it has capabilities to initialize the SPU with implicit and explicit completion.

Figure 5.12 shows the EPC process model with implicit completion of the SPU. The monitor shipment function receives shipment-monitoring events as input event stream. The function is supported by the monitor shipment ESPS (see Figure 5.11); it uses a capability of the ESPS to initialize an SPU with an implicit completion condition. The implicit completion condition is assigned to the connection between the EPC function and the ESPS *start processing with completion condition* capability. Implicit completion is triggered when the shipment arrives at its destination address.

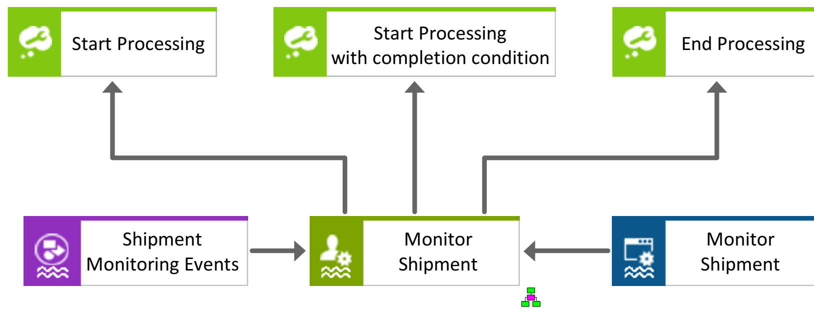


Figure 5.11: The shipment monitoring example Service Allocation Diagram.

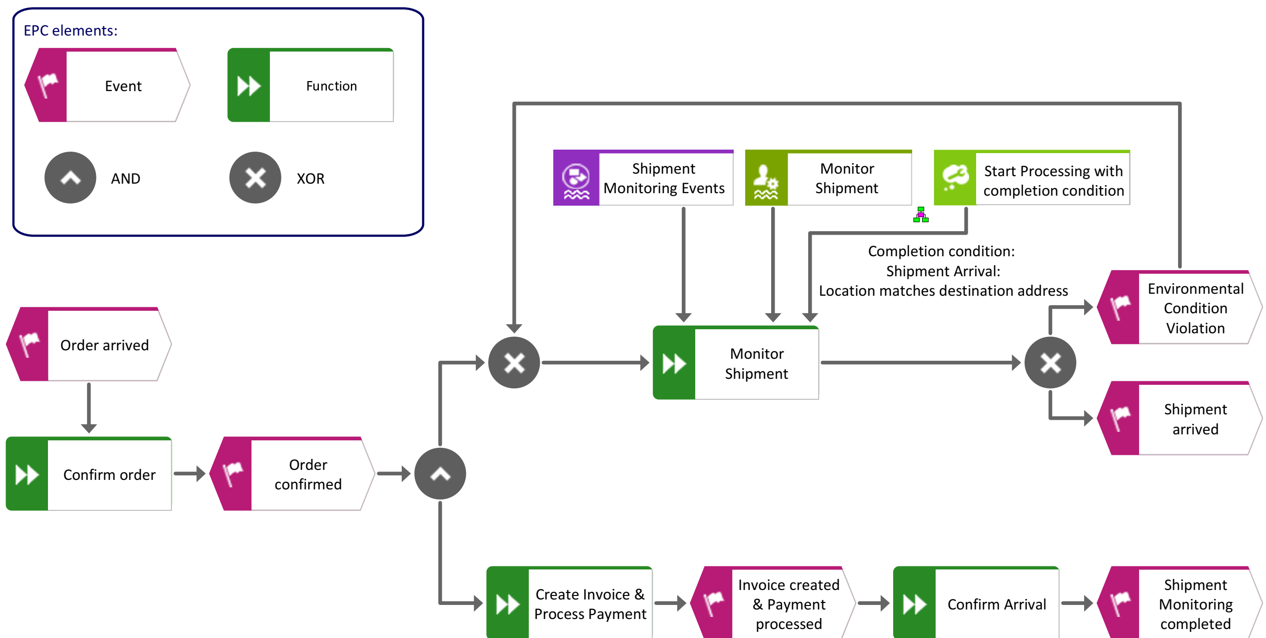


Figure 5.12: The shipment monitoring example business process as EPC utilizing an SPU with implicit completion

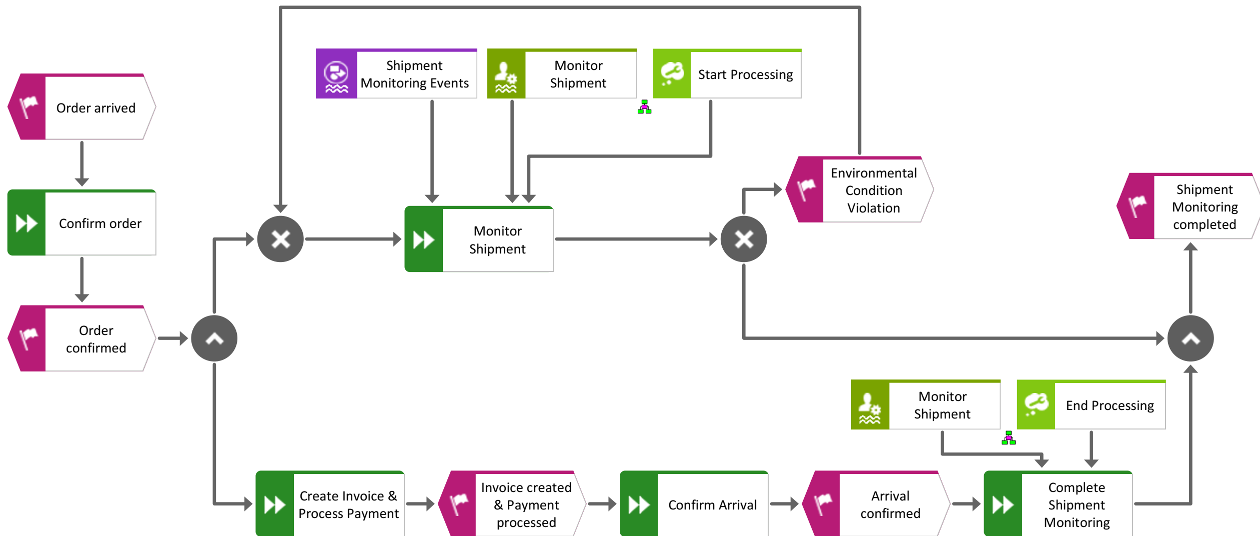


Figure 5.13: The shipment monitoring example business process as EPC utilizing an SPU with explicit completion

To report an environmental condition violation, the SPU is followed by an XOR operator: the process control flow triggers the environmental condition violation event and directly returns to the monitoring function. This control flow loop is instantaneous so that the SPU continues processing; the event stream processing is not interrupted at the technical layer. This SPU-specific loop pattern is used to model asynchronous messaging in EPCs as required in event stream processing scenarios. The completion of the SPU is triggered as soon as the completion condition is fulfilled, i.e., when the shipment reaches its destination. With completion of the SPU, the control flow moves on to the XOR operator and results in the shipment arrived event, which completes the process.

Figure 5.13 shows the order process with explicit completion of the SPU. The monitor shipment function is supported by the ESPS capability for initialization with explicit completion. An additional process step is added after the arrival confirmation to trigger the explicit completion. The *complete shipment monitoring* function uses the *end processing* capability of the shipment monitoring ESPS to trigger the completion of the shipment monitoring.

It is also possible to model SPUs with implicit and explicit completion in parallel. The monitor shipment function shown in Figure 5.12 is then combined with the complete shipment function shown in Figure 5.13.

Transformation from EPC to BPMN

EPCs are used for abstract process models from a business perspective; processes are modeled at the CIM layer. For the execution of processes a technical process representation is required that correlates with an executable process representation.

Since BPMN is a suitable notation for such technical models, EPC models can be transformed to BPMN in order to support automated process execution. This also requires a transformation of SPUs. Since SPUs are independent of a concrete modeling notation a mapping of SPUs in EPCs to SPUs in

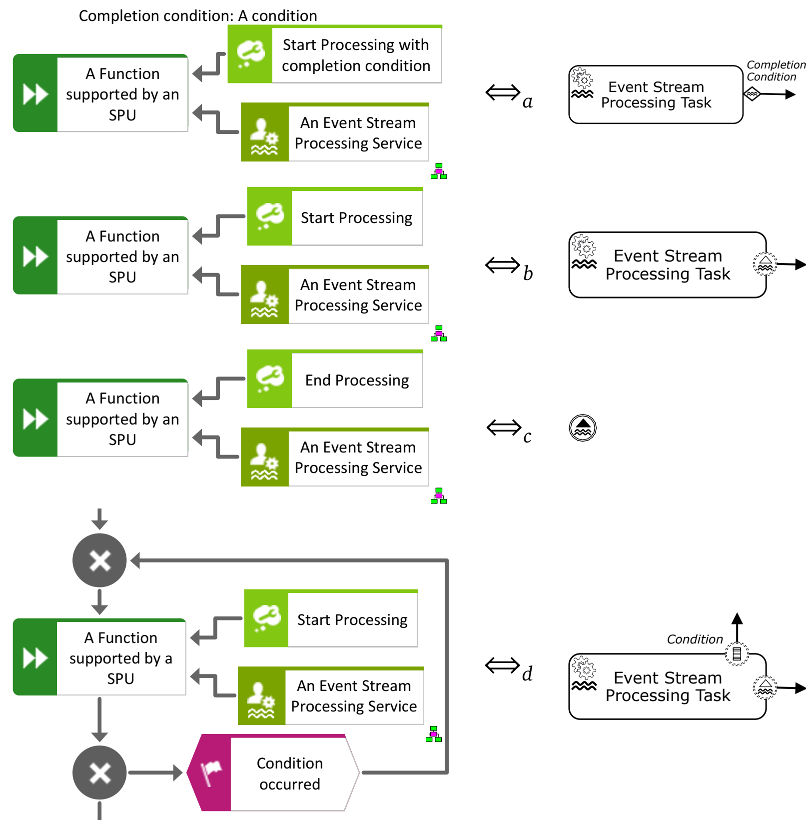


Figure 5.14: Mapping of SPUs between EPC and BPMN

BPMN is possible. The transformation from EPCs to BPMN can also be partly automated. This is, for example, supported by the Software AG ARIS business process platform.

Theoretical approaches for a mapping from EPCs to BPMN are given in [90]. The basic concept is the transformation of EPC functions into a BPMN tasks. EPC events are disregarded as long as they are not decision conditions for connectors. Operators, organizational elements, and data objects are directly transformed into their BPMN representations. In ARIS, for example, functions supported by a screen object are mapped to BPMN user tasks; functions assigned with an organizational object only are mapped to manual tasks. Analogously, we define the mapping from EPC functions that are supported by ESPSs to ESPTs in BPMN as shown in Figure 5.14.

An EPC function that is supported by an ESPS with implicit completion is mapped to an ESPT with implicit completion, i.e., the completion condition assigned to the connection between the EPC function and the capability is mapped to the outgoing sequence flow of the ESPT ($\Leftrightarrow_a$ in Figure 5.14). An EPC function that is supported by an ESPS with explicit completion is represented by an ESPT with explicit completion; the completion is triggered by an SPU intermediate boundary non-interrupting signal event ($\Leftrightarrow_b$ in Figure 5.14). This event is triggered by the SPU signal intermediate throwing event, which is the BPMN representation for the explicit completion capability used by an EPC function ($\Leftrightarrow_c$ in Figure 5.14). ESPTs in BPMN allow non-interrupting conditional events to be attached to the boundary of an ESPT for a conditional triggering of subsequent actions. In EPCs this is modeled with an event loop enclosing solely the event stream processing function ($\Leftrightarrow_d$ in Figure 5.14).

The function supported by an ESPS is not completed in this case; the control flow directly returns. The outgoing EPC event is consumed elsewhere and triggers concurrent functions.

Event streams in EPCs are mapped to their corresponding elements in BPMN: an EPC input ESS (*is input for* association) is mapped to the BPMN input ESS; an EPC output ESS (*has as output* association) is mapped to the BPMN output ESS. In EPCs the distinction between input and output event stream is based on the type of the association connection, in BPMN individual elements for both cases exist.

5.2.2 Workflow Execution Layer

The execution of business processes by an IT infrastructure requires a transition from the technical process model to an executable process format. The BPMN 2.0 standard itself specifies such execution semantics; the standard also provides examples for the mapping between BPMN and BPEL. Independent of the concrete technical representation format, the goal is to bridge the semantic gap between the technical model notation and interfaces of IT components so that the process can be executed automatically. The transition from a technical process model towards an executable process representation requires adding additional technical details.

For different task types and control flow components, execution languages provide executable representations. When the mapping of graphical process task and process control flow elements is complete and all necessary data is specified, the process execution engine is able to execute instances of the process. Each instance reflects a concrete business transaction, e.g., processing of Order No. 42. For each process instance, the execution engine orchestrates the different tasks, passes on task input and output data, and evaluates conditions specified in the control flow. Examples are the execution of BPMN service tasks and human tasks: a service task can be executed by calling a web service. For this, the execution engine needs the service address as well as the input data to send to a service and the format of the expected output data from this service. For the execution of human tasks, process execution engines typically provide a front end to perform the work necessary to complete the task.

At the execution layer we define the technical details that allow ESPTs to be mapped to IT components. The mapping mechanism has to take into consideration that events arrive indefinitely and are not known when the control flow reaches an ESPT. Thus, the data input must be specified as a *subscription* for desired events that arrive during the execution period of an ESPT. During process execution, this subscription has to partition the event stream in process instance specific sub streams: when a process instance is created for a certain business task, e.g., processing of Order No. 42, the event stream has to be partitioned in sub streams of events relevant for the different order process instances. This requires events to hold an attribute that allows an association with a process instance. Monitoring events contain, for example, a shipment ID. This is shown in Figure 5.15: a monitoring task must be active for each process instance. This task instance has to receive all monitoring events for the shipment that is handled in this process instance. Given that each event carries a shipment ID, each monitoring task instance can issue a subscription for the appropriate events using the shipment ID as filter. When the process instance ID correlates with the shipment ID, the subscription can also be derived by the process execution engine on the basis of the process instance ID.

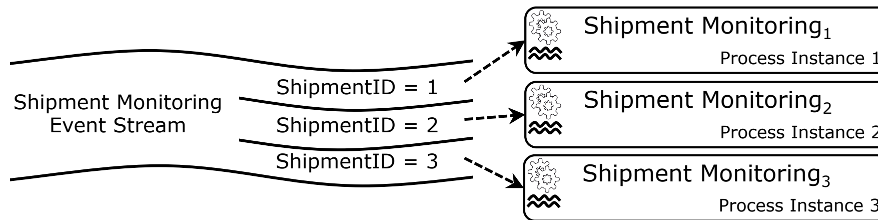


Figure 5.15: Process execution: Event Stream Processing Tasks (ESPTs) receive entity-centric sub streams of events.

The subscription parameters are essential for the instantiation of an ESPT. Like the input data passed on to a service during a service call, the subscription is part of the input data during an ESPT instantiation. Further, when the ESPT is modeled with an implicit completion, the completion condition is part of the input data required for the instantiation. As for ESPT completion, different ESPT instantiation strategies are possible. The push-based nature of stream processing allows an implicit creation of ESPT instances upon the arrival of appropriate events. In addition, ESPT instances can also be created explicitly by the process execution engine. When switching from explicit to implicit instantiation, the responsibility of instantiation moves from the process execution engine to the IT infrastructure. Implicit instantiation is useful when the moment of instantiation cannot be determined by the execution engine. It is also the more natural approach with respect to the characteristics of event streams; application logic is executed as soon as appropriate events are available. We support both instantiation schemes to allow for a high flexibility ($\rightarrow R_8$). Independent of the instantiation scheme, a subscription does not guarantee the availability of events, e.g., that events for Shipment No. 42 are published. Explicitly instantiated ESPTs can use a timeout to detect such an absence of events. With implicit instantiation, ESPT instances are not created in this case; the execution environment can detect and report this.

ESPT Instantiation

The execution of a business process leads to process instances that may run in parallel. Each ESPT in the model has corresponding ESPT instances that are created during process execution. Each ESPT instance processes the event streams relevant for a particular process instance (see Figure 5.15). The process execution engine can create an ESPT instance explicitly during the execution of a process instance. The subscription parameters required for the explicit instantiation must be derived per process instance; they define the sub stream of events that has to be processed by a particular ESPT instance, e.g., monitoring events for Shipment No. 42. The *explicit instantiation* is specified as follows ($\rightarrow R_6, R_7, R_8$):

```
1 EsptInstantiate(EsptName, EventStreamFilter, SubStreamAttribute,
2     SubStreamId [, CompletionCondition])
```

For the monitoring example, the explicit instantiation of a monitoring task for Shipment No. 42 without and with completion condition is:

```
1 EsptInstantiate(MonitorShipment, MonitoringEvent,
2     ShipmentId, 42)
```

```
1 EsptInstantiate(MonitorShipment, MonitoringEvent,
2     ShipmentId, 42, destination.equals(location))
```

An ESPT is referenced by name: `EsptName`, e.g., `MonitorShipment`. The subscription parameter has three parts: First, a general filter for events of interest that applies to all instances of an ESPT is specified as `EventStreamFilter`, e.g., monitoring events. Second, the `SubStreamAttribute` defines the part of the event data that partitions the event stream with respect to ESPT instances, e.g., the shipment ID; both are static expressions and derived based upon the ESS used in the model. Third, the `SubStreamId` defines the concrete event sub stream for which an ESPT instance should be created, e.g., Shipment No. 42. The `SubStreamId` is dynamic and derived per process instance by the execution engine at run time, e.g., based on the process instance ID. The optional `CompletionCondition` can be specified for implicit completion, e.g., defining a time out.

With implicit instantiation, the process execution engine only registers a static subscription pattern for an ESPT once, e.g., with the registration of the process. Since events arise in a push-style manner, the IT infrastructure is able to create ESPT instances implicitly at run time. The *implicit instantiation* is specified as follows ($\rightarrow R_6, R_7, R_8$):

```
1 EsptRegister(EsptName, EventStreamFilter,
2     SubStreamAttribute [, CompletionCondition])
```

For the shipment monitoring example, the ESPT registration is:

```
1 EsptRegister(MonitorShipment, MonitoringEvent, ShipmentId)
```

In contrast to explicit instantiation, the execution engine is not responsible for the dynamic subscription part anymore. Rather, the IT infrastructure ensures, that an ESPT instance is created for each distinct value of the `SubStreamAttribute`, e.g., for each shipment ID.

Upon implicit instantiation, the creation of ESPT instances is not synchronized with the control flow of the process execution. ESPT instances are created based upon the availability of events, i.e., as soon as events for a certain entity instance are available the corresponding ESPT instance is created. The availability of events for Shipment No. 42, for example, results directly in the creation of an ESPT instance that processes these events. This happens independently of the control flow of the process execution. In cases where this behavior is not desired, a synchronization step has to be performed between the beginning of the actual event processing and the control flow of the process execution. Two cases have to be taken into consideration: First, the process control flow reaches an ESPT and no ESPT instance has been created yet. Second, an ESPT instance is created although the control flow has not reached the ESPT.

In the first case, the process execution blocks and proceeds after the completion of an ESPT instance; this behavior is equivalent to the execution semantics of, e.g., service tasks. However, the process execution engine has no control over the instantiation and thus relies on the IT infrastructure. In the second case, the already created ESPT instance has to wait for the control flow of the process execution engine. This can be achieved by implementing a lock after the creation of an ESPT instance; this lock is released upon a dedicated signal from the process execution engine. Since a dedicated signal from the process execution engine is required, the explicit creation of ESPT instances is an alternative in such cases.

ESPT Completion

For the explicit completion of an ESPT instance, the process execution engine has to advise the IT infrastructure to perform a shutdown of particular ESPT instances, e.g., the shipment monitoring of Shipment No. 42. The completion command is specified as follows ($\rightarrow R_8$):

```
1 EsptComplete(EsptName, SubStreamId)
```

The SubStreamId identifies the ESPT instance that should be completed. In the monitoring example for Shipment No. 42, the following completion command is issued after the arrival confirmation task (cf. Figure 5.4):

```
1 EsptComplete(MonitorShipment, 42)
```

We distinguish between the control commands to manage ESPTs and the ESPT execution semantics. The control commands to register, instantiate, and complete ESPTs follow a request/reply pattern. Thus, our integration approach of event streams with business processes can be mapped to web service invocations. Web service invocation capabilities are part of most process execution engines so that ESPTs can be registered, instantiated, or completed; the ESPT name as well as further subscription and completion parameters are specified as variables in the service invocation. In addition to service invocation mechanisms, it might be necessary to implement a back channel for control flow purposes. Implicitly completing ESPT instances might have to notify the process execution engine about completion. This is the case when the control flow waits for a completion of an ESPT, e.g., when an ESPT is used before a BPMN AND-Join.

ESPTs with Output ESSs

ESSs used as output of ESPTs can be mapped to advertisements. Advertisements inform the IT infrastructure about the event types published by event producers; they allow the validation of subscriptions, i.e., whether events are potentially available for an issued subscription. The validation of subscriptions allows the identification of inconsistencies between demand and availability of event streams upon registration of ESPTs. Subscription validation by the process execution engine is only applicable when advertisements are mandatory for all publishers. In this case subscriptions are only valid when they are specified on event content, e.g., attributes, that is advertised by at least one publisher. In case not all event publishers are part of business process models, validating subscriptions against advertisements inside the process execution engine is not feasible; then only the IT infrastructure has a holistic view of event types, has to perform subscription validation, and informs the process execution engine.

The event types for the advertisements are derived from output ESSs and included as additional parameter at the instantiation or registration of ESPTs:

```
1 EsptInstantiate(EsptName, EventStreamFilter, SubStreamAttribute,
2     SubStreamId [, CompletionCondition] [, PublishedEventType])
```

```
1 EsptRegister(EsptName, EventStreamFilter,
2     SubStreamAttribute [, CompletionCondition] [, PublishedEventType
        ])
```

A *shipment monitoring SPU* with implicit completion that outputs *threshold exceeded events* (cf. Figure 5.6) is then registered as follows:

```
1 EsptRegister(MonitorShipment, MonitoringEvent, ShipmentId,
2     destination.equals(location), ThresholdExceededEvent)
```

When the *report threshold violation SPU* is registered, the IT infrastructure knows that the demanded `ThresholdExceededEvent` type is available:

```
1 EsptRegister(ReportExceededThreshold, ThresholdExceededEvent,
2     ShipmentId)
```

ESPT Mapping in BPEL

Business process models that contain ESPTs can be mapped to BPEL. However, the BPEL standard [127] does not support all concepts required for a complete mapping of the different instantiation and completion strategies. ESPTs with explicit instantiation and explicit completion can be mapped to standard BPEL: the explicit instantiation is realized as web service call. The return from this call is blocked by the IT infrastructure until the ESPT instance is explicitly stopped by an `EsptComplete` service invocation. Explicit instantiation and completion in BPEL are as follows:

```
1 <invoke partnerLink="EsptWebService" operation="EsptInstantiate"
2     inputVariable="explicitInstantiateParams"
3     outputVariable="completed"/>
```

```
1 <invoke partnerLink="EsptWebService" operation="EsptComplete"
2     inputVariable="explicitCompletionParams"/>
```

With implicit instantiation, single ESPT instances are transparent to the process execution engine. The registration of ESPTs has to be performed once with the registration of a process; the ESPT instances are then created automatically. The BPEL standard does not support hooks for service invocation upon the registration of new processes. Thus, a BPEL execution engine has to be extended with these capabilities to support implicit instantiation of ESPTs. The hook for execution at process registration can be part of the BPEL code itself; when a new process is registered and checked, this part of the process is executed only once:

```
1 <atRegistration><invoke partnerLink="EsptWebService" operation=
2     "EsptRegister" inputVariable="implicitInstantiateParams"/>
3     </atRegistration>
```

When an ESPT is invoked implicitly, there is no BPEL web service invocation in each process instance. Thus, a blocking service invocation cannot be used to interrupt the control flow until completion of an ESPT instance. Rather, the process execution engine has to be notified externally about the completion of an ESPT instance so that the control flow can proceed. Extensions to BPEL engines to react on such external triggers have been proposed, e.g., in [97] and [95]. The ESPT can be mapped to a barrier that is released when the ESPT instance signals its completion.

5.2.3 IT Infrastructure Layer

A major goal for BPM is the seamless execution of business process models inside an IT infrastructure with minimal transition effort. Business functions are mapped to human tasks or service tasks. These tasks are executed by a process execution engine; human tasks are executed with a workflow management system, service tasks are mapped to services inside a SOA.

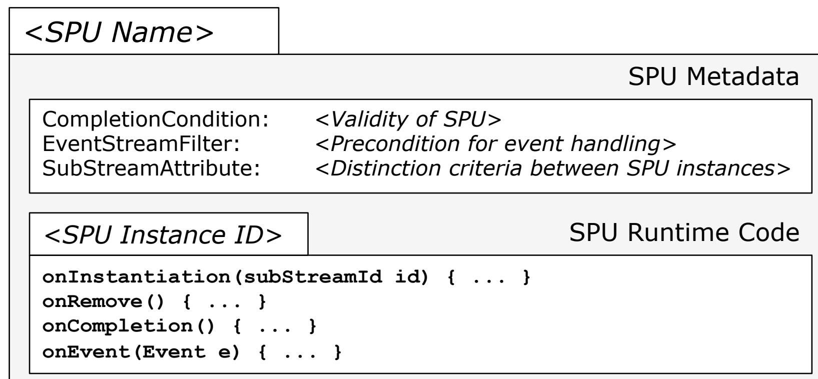


Figure 5.16: SPU structure: SPU metadata and SPU runtime methods.

For event stream processing, SPUs are a suitable model to represent ESPTs at the IT infrastructure layer. SPUs encapsulate event stream processing logic with respect to a certain entity, e.g., shipments ($\rightarrow R_{10}$). An SPU instance subscribes to events of a certain entity instance, e.g., Shipment No. 42 ($\rightarrow R_{11}$). Figure 5.16 shows the SPU structure; we adapted the nomenclature to the specific use case of event stream integration with business processes. The grouping attribute to define the sub stream of events associated with a certain entity instance is specified as Sub Stream Attribute[1] in the SPU metadata, e.g., the shipment ID. Further, the metadata holds the Completion Condition[2], e.g., a timeout, as well as the Event Stream Filter[3] as a general subscription filter applied by all SPU instances, e.g., monitoring event. SPU instances are created implicitly or explicitly ($\rightarrow R_9$). With implicit instantiation the middleware ensures that an SPU instance is active for each distinct value of the sub stream attribute, e.g., for each shipment in transport. With explicit instantiation, SPU instances are created manually by specifying a concrete sub stream attribute value, e.g., Shipment No. 42. The completion of SPU instances is triggered implicitly by the completion condition, or explicitly by a command ($\rightarrow R_9$). SPU instances run in a distributed setting and have a managed lifecycle; application logic can be executed upon instantiation, removal, completion, and upon event arrival ($\rightarrow R_{11}$). The runtime environment for SPUs, which we present in Chapter 6, provides a web service interface and supports the commands introduced in Section 5.2.2. This allows the execution of SPUs by business process execution engines.

5.3 Implementation

We implemented SPU modeling capabilities for EPCs and BPMN in Software AG's ARIS platform. ARIS is a business process platform for business process analysis, enterprise architecture, and gov-

[1] Referred to as Instantiation Expression in the generic SPU model in Section 3.2.3.
[2] Referred to as Validity Expression in the generic SPU model in Section 3.2.3.
[3] Referred to as Constant Expression in the generic SPU model in Section 3.2.3.

ernance, risk & compliance. ARIS supports modeling of processes with EPCs as well as with BPMN. For our implementation we used ARIS Design Server 9.0 and the ARIS Architect 9.0. To support SPU modeling in EPCs and BPMN within ARIS, we added new notation element symbol types and connection attributes for EPCs (see Figure 5.9) and BPMN (see Figure 5.3) to the configuration of the ARIS server; the ARIS server acts as central repository for process models and process model components.

For EPCs we added symbol types for:

- a cluster/data model object (derived from cluster symbol type) to model ESSs;
- a service type object (derived from business service symbol type) to model ESPSs; and
- an application system type object (derived from software service symbol type) to model the technical representation of ESPSs.

For BPMN we added symbol types for:

- two cluster/data model types (derived from data input/output symbol types) to model ESSs;
- a function type object (derived from service task symbol type) to model ESPTs; and
- two event object types (derived from signal intermediate event symbol type) to model SPU signal intermediate events (throwing and non-interrupting) required for explicit completion.

Further, we added an attribute type and an attribute type symbol for BPMN to model the condition for implicit completion.

With the added symbol types, attribute symbol types, and attribute types it is possible to model SPUs and event streams in EPC and BPMN diagrams. All new modeling elements are usable in different diagram types as the symbols they are derived from. They can be used, for example, in application system type diagrams, all types of EPCs, function allocation diagrams, and service allocation diagrams.

Process Model Execution

One goal of BPM is to enable the automated execution of business process models. In Section 2.2.2 we introduce the different layers in such a model-to-execute (M2E) process and show that a high coherence across the process modeling layer, the process execution layer, and IT infrastructure layer is necessary. Our approach of SPUs is a mechanism to encapsulate event stream processing to achieve such a high coherence and to provide the foundation for M2E.

Our extensions to EPCs allow business experts to create abstract business process models that contain SPUs; our extensions to BPMN along with the proposed EPC-to-BPMN mapping allow the transformation of abstract EPC models to technical BPMN models. These BPMN models are further refined during the M2E process and brought to execution. The execution as such is detailed in Section 5.2.2 where the mapping between ESPTs and service invocations is presented.

We are working with Software AG on the implementation of M2E mechanisms that support SPUs. In the following we illustrate the M2E workflow and show how SPU integration is achieved. Our M2E approach is based upon the Software AG ARIS, CentraSite and webMethods product suites. Business

process analysis and process modeling is supported by the ARIS Architect; it provides an integrated platform where process models are created and governed collaboratively. At the beginning of the M2E workflow EPC process models are created with the ARIS Architect; these models reflect an abstract business perspective. In the next step a transformation process is applied; abstract EPC process models are mapped into a logical process model represented with BPMN. In ongoing work we are working on the customization of the ARIS model transformation framework, which performs the EPC-to-BPMN mapping. We are integrating the mapping of ESPSs in EPCs to ESPTs in BPMN. This allows a partly automated transformation from EPC models with ESPSs to BPMN models.

After the transformation the resulting BPMN model needs to be refined by a process engineer. This involves the adaptation to technical concepts and restrictions, e.g., adapting events for inter and intra process communication and error handling scenarios. Finally, changed process elements are linked to the related elements in the original EPC model using the process alignment capability of ARIS. This allows a synchronization of the process between abstract and technical layer; refinements in the EPC model are applied - if possible automatically - to the BPMN model and vice versa.

The next step in the M2E workflow is the transition from a technical model in ARIS to an executable process representation that is deployed to a process execution system. For SPUs this involves linking the Eventlet middleware via its web service interface to ESPTs of the BPMN model (see Section 5.2.2). In Software AG's M2E the executable process is created with the Software AG Designer. Process developers import the technical process model from ARIS in the Software AG Designer. In the Designer the process is represented as a technical BPMN diagram and technical implementations are mapped to process steps. This is supported by the Software AG CentraSite service repository: the web service interface of our Eventlet middleware is registered with CentraSite. This allows an easy assignment of the Eventlet service (including invocation parameters) to ESPTs in the model. Currently, only explicit instantiation and completion of SPUs is supported. Implicit instantiation would require a service invocation at process registration, implicit completion would require a feedback mechanism from the Eventlet middleware to the process execution environment.

Processes are synchronized between the webMethods platform and the ARIS platform. Changes to technical processes are propagated to the abstract process models and vice versa. During this round tripping, approval steps ensure that processes at the IT level and at the business level remain synchronized. The executable process model is deployed to the webMethods Integration Server where the individual process steps are executed by the different components of the webMethods BPMS; SPUs are executed by the Eventlet middleware controlled by web service invocations from webMethods BPMS.

5.4 Summary

In collaboration with Software AG, we identified the need to integrate event streams with business process modeling and execution. Rather than single events, event streams are considered as business relevant units in this context. We developed a generic integration at the process modeling, the process execution, and the IT infrastructure layer. We use SPUs as abstraction mechanism for event stream processing and specify BPMN and EPC process modeling notation extensions that reflect SPUs. Like services, SPUs can be used to encapsulate business functions with event streams as business relevant data source; this allows intuitive modeling from the business perspective. We take

semantics of event processing into account and support implicit as well as explicit instantiation and completion strategies. The abstraction paradigm of SPUs leads to a high coherence across the layers. This minimizes the transition overhead from the graphical model notation, to the executable process description, to the IT infrastructure. Our approach is a clear separation of concerns; ESPTs and ESPSs are declarative, the (imperative) application logic resides solely at the technical layer inside SPUs. With our approach, widely adopted event stream processing techniques, like CEP, can be encapsulated; this makes event stream processing available coherently across the business process modeling, the business process execution, and the IT infrastructure layer.

6 Eventlets: An Implementation of SPUs

Event Stream Processing Units (SPUs), as containers for event stream processing logic, require a runtime infrastructure for execution. We introduce the structure and runtime semantics of SPUs in Section 3.2. The core of our model is support for implicit and explicit instantiation and completion. In this chapter we describe our middleware that implements the SPU concept and provides a runtime infrastructure for SPU instance creation and execution. We refer to the technical representation of an SPU as *Eventlet* - short for *event applet*. We make this distinction to indicate that our Eventlet middleware is one possible technical representation of the SPU container model; like web services are one possibility to implement a Service-oriented Architecture (SOA), Eventlets are one realization of the SPU container model.

We first introduce the architecture of our Eventlet middleware; this involves the different components required to implement the runtime semantics of SPUs. We then present our Java Enterprise based implementation in detail. In previous work, we identified that data heterogeneity is a challenge with respect to event-based systems; to overcome this issue we show how our transformation approach presented in [73, 74] is applicable for Eventlets. We also discuss security aspects of Eventlets and apply our privacy preserving pub/sub approach presented in [120,121] to Eventlets.

Our Eventlet middleware supports implicit and explicit creation and completion of Eventlet instances. This involves a dynamic derivation of subscriptions based upon constant and instantiation expressions. Eventlet instances are manageable and follow a certain lifecycle; the Event middleware architecture is designed for scalability: Eventlet instances can be distributed across multiple machines.

6.1 Eventlet Middleware Architecture

Our Eventlet middleware is the runtime infrastructure for Eventlets which act as containers for event stream processing application logic. Developers implement *Eventlet Prototypes* that follow the structure of an SPU. Eventlet prototypes contain generic event stream processing application logic along with metadata required for the creation of entity-instance specific Eventlet instances. Eventlet prototypes are registered with the middleware, which creates an Eventlet monitor for each prototype. The Eventlet monitor is responsible for the creation and management of Eventlet instances. The Eventlet middleware functionality is transparent to the developer who only has to register Eventlet prototypes. We first introduce the different components of the Eventlet middleware and then describe the runtime behavior.

6.1.1 Eventlet Middleware Components

The Eventlet middleware architecture along with relations amongst their components is depicted in Figure 6.1. It consists of three layers:

- Control layer;
- Instantiation layer; and
- Runtime layer.

The control layer contains components that expose an interface to interact with the Eventlet middleware. At the instantiation layer Eventlet monitors are responsible for the creation of Eventlet instances from registered prototypes. Eventlet instances are then executed at the runtime layer where the actual event stream processing is performed. The components are described in more detail in the following.

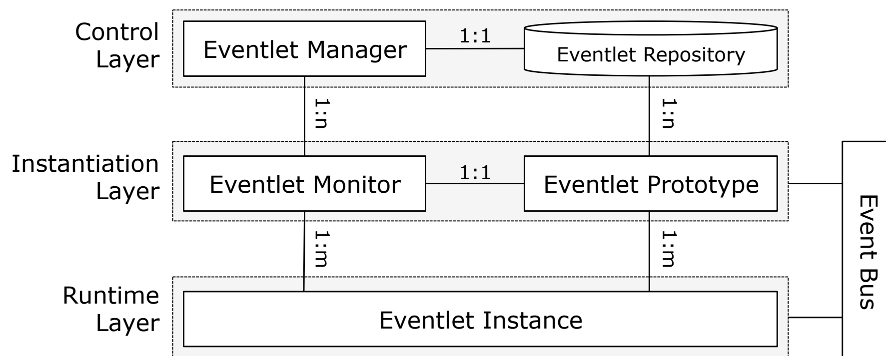


Figure 6.1: Eventlet middleware components: Eventlet instance are created and controlled by Eventlet monitors.

Event Bus

The event bus plays a crucial role in the conceptual design of an Eventlet middleware. It is the core mechanism to transport events from event producers to Eventlet instances. As described in Section 3.1 the event bus follows the pub/sub paradigm to decouple all components [132]. Pub/sub systems support different subscription mechanisms as described in Section 2.1.1. To implement the SPU container model, a content-based subscription mechanism is the best fit since the constant and instantiation expressions of SPUs are defined on event content. However, the Eventlet middleware can be adapted to support subscription mechanisms that do not provide full content-based capabilities.

Eventlet Manager

The Eventlet manager is the main component of our Eventlet middleware. It provides an interface to the user for the registration and management of Eventlet prototypes. It also provides an interface to

external applications; for the use of Eventlets with business process execution engines, the Eventlet manager provides a web service interface.

Eventlet Monitor

Eventlet monitors are responsible for monitoring events and instantiating new Eventlets; they ensure that the required execution semantics of SPUs are fulfilled. For this, one Eventlet monitor is associated with each Eventlet prototype. Eventlet monitors implement implicit and explicit instantiation of Eventlets as specified in Section 3.2.

Eventlet Prototype

Eventlet prototypes are the implementation of SPUs. They are containers for entity-instance-centric event stream processing logic and follow the structure presented in Section 3.2.3: Eventlet prototypes contain meta data required for the creation of Eventlet instances (constant expression, instantiation expression, and validity expression) as well as runtime code to implement the actual event stream processing; this encompasses at least the implementation of the `onEvent` method.

Eventlet Repository

The Eventlet repository is a middleware component to hold Eventlet prototypes and their metadata. The repository allows the Eventlet manager to create Eventlet monitors and Eventlet instances on different nodes. The repository can be replicated for scalability.

6.1.2 Runtime Behavior

At runtime, the different Eventlet middleware components interact to provide a suitable runtime environment for Eventlet instances. The runtime behavior can be partitioned in several steps as shown in Figure 6.2. After starting up the Eventlet manager, the Eventlet middleware accepts requests. Developers implement an Eventlet prototype corresponding to the structure shown in Figure 3.3 (Step 1). This prototype is then registered with the Eventlet middleware (Step 2). Upon registration of the prototype, the Eventlet manager stores the prototype in the repository (Step 3) and triggers the creation of an Eventlet monitor associated with the Eventlet prototype (Step 4). This Eventlet monitor is then responsible to create Eventlet instances. Depending on the instantiation strategy the Eventlet monitor behavior differs; with implicit instantiation the Eventlet monitor triggers the creation of Eventlet instances actively, with explicit instantiation it waits for external commands. In the following we describe Steps 4, 5, and 6 in more detail; these steps depict the core functionality of the Eventlet middleware. We use our shipment monitoring example and describe a shipment monitoring Eventlet to illustrate the runtime behavior of our middleware.

Example: Shipment Monitoring Eventlet

With our SPU model, the task of shipment monitoring is encapsulated in an entity-type-centric way. The required metadata needs to be provided by developers. An Eventlet prototype that implements

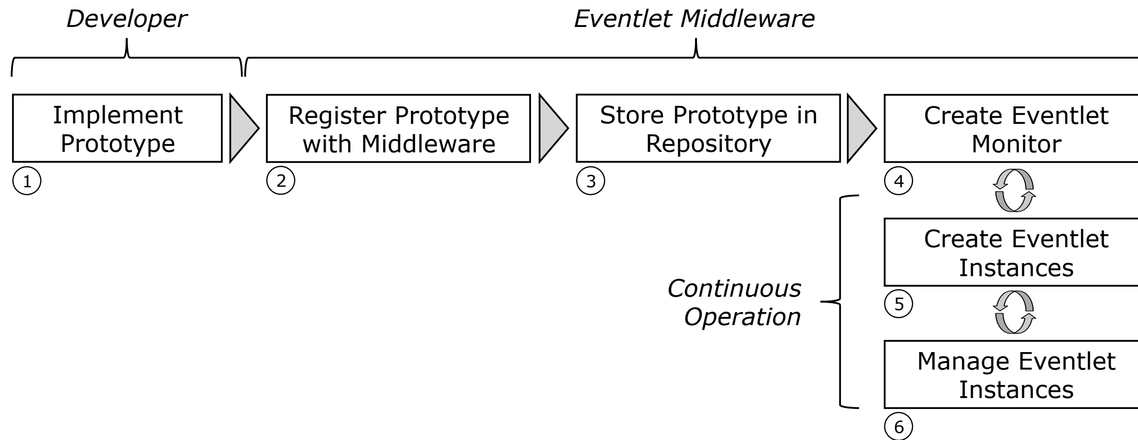


Figure 6.2: Eventlet development and deployment workflow; after deployment, Eventlet monitors continuously watch the event stream and trigger Eventlet instantiations.

a monitoring task has to specify a constant expression, an instantiation expression, as well as a validity expression. For the monitoring of a shipment, the Eventlet prototype metadata can be as follows (assuming events in att/val representation):

```
1 constantExpression := "type == monitoringEvent";
2 instantiationExpression := "shipmentID";
3 validityExpression := "timeout(120sec)";
```

When an Eventlet prototype with this metadata is registered with the Eventlet middleware and an implicit instantiation strategy is applied, the middleware ensures that an Eventlet instance is created for each shipment of which monitoring events are published.

Eventlet Monitor Operation

Eventlet instances execute the application logic specified in the Eventlet prototype. Each Eventlet instance subscribes to events associated with a certain entity instance and provides a container with local state that encapsulates application logic. As specified in the SPU model, the creation of Eventlet instances can be triggered implicitly or explicitly (cf. Section 3.2.2).

Implicit instantiation ensures that Eventlet instances are created automatically as soon as events of a particular type are published. For this, an Eventlet monitor subscribes to all events that match the constant expression of the associated Eventlet prototype. This Eventlet monitor is created upon registration of an Eventlet prototype with the Eventlet Manager. The Eventlet monitor issues a subscription derived from the constant expression of the Eventlet prototype. The constant expression specifies the event stream of interest for which Eventlet instances have to be created. The subscription operation is as follows:

```
1 subscribe(constantExpression);
```

For the monitoring example, the subscription is:

```
1 subscribe("type == monitoringEvent");
```

With this subscription the Eventlet monitor receives all events for which an Eventlet instance is active or needs to be created. For incoming events, the Eventlet monitor evaluates the instantiation expression: when no Eventlet instance exists for a specific instantiation value set, the Eventlet monitor triggers the instantiation. This is shown in Algorithm 1: the instantiation expression of the Eventlet prototype is applied to each event that arrives at the associated Eventlet monitor. The resulting set of attribute values is used as input for a lookup to check whether this instantiation value set has already been observed before. If this is the case, an Eventlet instance is already active and processes the events. If this is not the case, the creation of an Eventlet instance is triggered and the retrieved instantiation value set is used as input parameter. The instantiation value set is also added to a list so that further lookups for this instantiation value set succeed and no new Eventlet instances are created.

Data: list $activeEventletInstances$ that contains instantiation value sets
Data: events e matching constant expression
while *events arrive* **do**
  retrieve attribute values $instVal$ by applying instantiation expression to arriving event e;
  if *$instVal$ is not in list $activeEventletInstances$* **then**
    create Eventlet instance with $instVal$ as instantiation parameter;
    add $instVal$ to $activeEventletInstances$ list;
  end
end

Algorithm 1: Eventlet monitor for implicit instantiation

In our example of an Eventlet for shipment monitoring, the instantiation expression specifies the `shipmentID`. An Eventlet monitor for implicit instantiation checks for each event of type `monitoringEvents` (as specified in the constant expression) whether the value of the attribute `shipmentID` is already contained in the list of active Eventlet instances. If not, the Eventlet monitor triggers an instantiation. A more complex instantiation expression might specify the `shipmentID` in combination with the `truckID` as identifier for an entity instance specific event stream. In this case the Eventlet monitor checks whether the conjunction of both attribute values is already contained in the list of active instances, e.g., whether `42;37`, which stands for Shipment No. 42 and Truck No. 37, is present.

With *explicit instantiation* the full control of instantiation is left to the users, i.e., the creation of Eventlet instances must be requested explicitly. This is applicable in scenarios where external knowledge about the entity instances for which event stream processing should happen is available, e.g., knowledge about shipments currently in transport. Explicit instantiation is also helpful for debugging purposes to manually test the implementation of Eventlet prototypes. For explicit instantiation the instantiation set values for which an Eventlet instance should be created have to be provided externally. An instantiation request is then sent to the Eventlet middleware, e.g., to create an Eventlet instance for the monitoring of Shipment No. 42. As for implicit instantiation, an Eventlet monitor is responsible for the explicit instantiation of Eventlets. The creation of the Eventlet monitor is also triggered with the registration of an Eventlet prototype. However, in case of explicit instantiation the Eventlet monitor does not issue a constant expression based subscription to monitor the event stream. The Eventlet monitor only keeps track of active Eventlet instances; this allows the Eventlet monitor to prevent the creation of duplicate Eventlet instances, i.e., of multiple instances that per-

form event stream processing for events of the same entity instance. The explicit instantiation logic is shown in Algorithm 2.

Data: list $activeEventletInstances$ that contains instantiation value sets
Data: instantiation request with instantiation value set $instVal$
if $instVal$ *is not in list* $activeEventletInstances$ **then**
  create Eventlet instance with $instVal$ as instantiation parameter;
  add $instVal$ to $activeEventletInstances$ list;
end

Algorithm 2: Eventlet monitor for supervised explicit instantiation

Eventlet monitors are also involved in the completion process of Eventlet instances. When Eventlet instances complete, the Eventlet monitor is informed about this completion to update the list of active Eventlet instances. There are two options: marking the instantiation value set as inactive or removing it from the list. When the instantiation value set of a completed Eventlet instance is removed from the list, an Eventlet instance for the processing of this instantiation value set is created at the next occurrence. For example, the Eventlet instance for monitoring Shipment No. 42 completes due to a timeout. The Eventlet monitor then removes Shipment No. 42 from its list of active Eventlet instances; the next time an event of Shipment No. 42 arrives, a new Eventlet instance is created. When an instantiation value set is marked as inactive, further occurrences of events for this instantiation value set do not trigger the instantiation of an Eventlet, i.e., although events of Shipment No. 42 arrive, the inactive marked instantiation value 42 prevents the creation of a new Eventlet instance.

The explicit completion of Eventlet instances is handled by the Eventlet monitor. The Eventlet monitor receives the request for shutdown of a particular Eventlet instance identified by its instantiation value set, e.g., shutdown request for the Eventlet instance that performs monitoring of Shipment No. 42. As for implicit completion, there are the options to mark the instantiation value set as inactive, to prevent re-instantiation, or to remove the instantiation value set from the list of active instances. The Eventlet monitor then shuts down the Eventlet instance.

Eventlet Instance Creation

The creation of Eventlet instances is triggered by Eventlet monitors. The Eventlet monitor uses the Eventlet prototype as blueprint and creates an instance for the processing of entity-instance-centric events identified by an instantiation value set. The Eventlet monitor creates the Eventlet instance and uses this instantiation value set as parameter. In combination with the Eventlet prototype metadata an Eventlet instance is now able to issue its subscription. It combines the constant expression, the instantiation expression, and the instantiation value set into a single filter expression to receive events related to a single entity instance (cf. Section 3.2.4):

```
1 subscribe(constantExpression AND (instantiationExpression EQUALS
    instantiationValueSet));
```

For the monitoring example with shipmentID as instantiation expression and an instantiation value of 42 (Shipment ID) this results in:

```
1  subscribe("type == monitoringEvent" AND "shipmentID == 42");
```

The more complex instantiation value set 42;37 (Shipment No. 42, Truck No. 37), results in the following subscription that is issued at the creation of the Eventlet instance:

```
1  subscribe("type == monitoringEvent" AND ("shipmentID == 42" AND "
      truckID == 37"));
```

Prior to issuing the subscription, the (optional) onInstantiation method is invoked. In this method, initialization of components required for the event stream processing can be performed. After the subscription is issued the onEvent method is invoked asynchronously at incoming events; by postponing the subscription until the onInstantiation method has been completed, it is ensured that the Eventlet instance is prepared for the processing of arriving events. Mechanisms for the handling of events that arrive during this instantiation process are discussed in the context of future research in Section 10.2.

Each Eventlet instance is autonomous, i.e., there is no direct connection with the Eventlet monitor. At the technical layer Eventlet instances establish their own connections with the event bus and receive events according to their subscription. The Eventlet monitor has knowledge about active Eventlet instances and can control them; the event stream processing as such is independent of the Eventlet monitor. This is illustrated in the runtime view shown in Figure 6.3: Eventlet instances created by the Eventlet monitors run within the Eventlet runtime environment. This environment for Eventlet instances provides functionalities for Eventlet instance monitoring and management. The runtime environment is also responsible to distribute Eventlet instances across multiple machines.

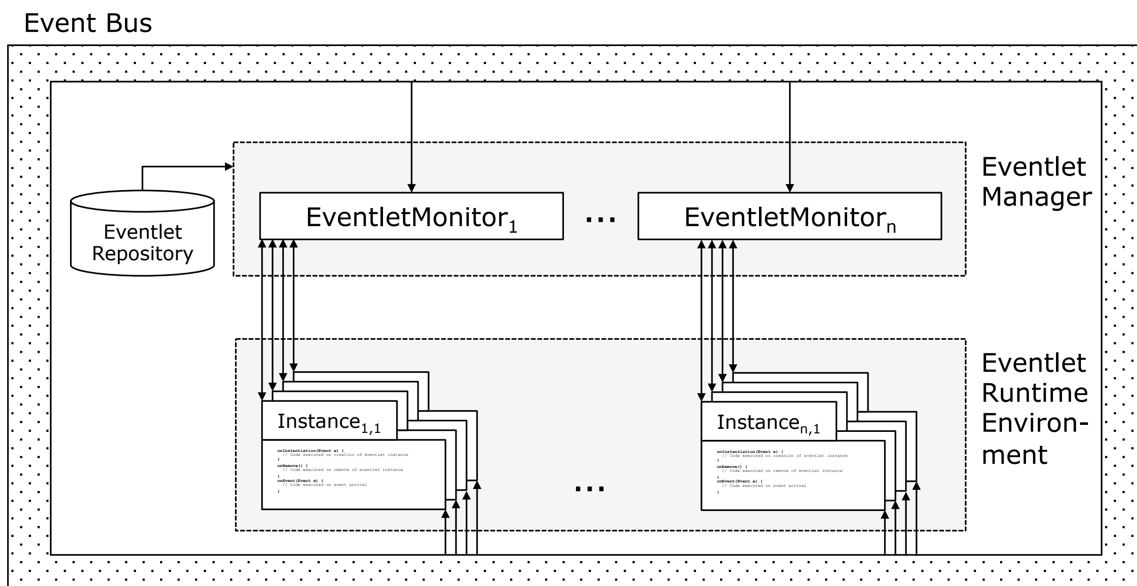


Figure 6.3: Eventlet middleware runtime view: Eventlet monitors know the Eventlet instances they are responsible for. Eventlet instances have dedicated connections to the event bus.

Eventlet Instance Expiration

Each Eventlet instance evaluates its validity expression during runtime to decide whether the validity condition is violated. This evaluation happens transparently, i.e., it does not have to be implemented explicitly. Depending on the actual validity expression, the evaluation is, for example, triggered by a timer or by an incoming event. The support for different kinds of validity expressions requires an according implementation within the Eventlet middleware. In case of a validity condition violation the `onExpiration` method is called to handle the expiration. It is possible to resume the normal mode of operation of the Eventlet instances by resetting the expiration. In case of a timeout, for example, the timer is reset. Since Eventlet instances are stateful, it is possible to keep track of validity expression violation and react, for example, only when a timeout occurs repeatedly. The validity expression is also the mechanism to realize an implicit completion of Eventlet instances. It is applied in our approach for the integration of event stream processing with business processes: the completion condition specified in the process models results in a validity expression of an Eventlet prototype (see Section 5.2.3).

Eventlet Instance Lifecycle

Eventlet instances follow a lifecycle as shown in Figure 6.4. Derived from its Eventlet prototype, the lifecycle of an Eventlet instance begins with its instantiation triggered by the Eventlet monitor. During the initialization the subscription filter is derived and the `onInstantiation` method is called. With the completion of the initialization, the subscription is issued and the Eventlet instance is in the active state. Events arriving while the Eventlet is in the active state trigger the `onEvent` method and events are processed. An Eventlet instance in active state can be paused, can expire, or can be removed. When paused, the Eventlet instance does not receive new events, i.e., the subscription is suspended. Meanwhile, the Eventlet instance remains active and keeps state. When resumed, the subscription is reissued. As described above, the violation of the validity expression leads to the invocation of the `onExpiration` method and brings an Eventlet instance to the expired state. The normal operation can then be resumed, the Eventlet instance can be removed or paused depending on the use case. When an Eventlet instance is removed, all state is lost. Prior to removal, the `onRemove` method is invoked, for example, to persist data. Upon removal the instantiation value set of an Eventlet instance can be marked as inactive; in this case no Eventlet instance is created for this particular instantiation value set although matching events are observed at a later point in time. The default behavior is a removal where the instantiation value set is also removed from the Eventlet monitor's list of active instances; this allows the repeated creation of Eventlet instances for the same instantiation value sets.

6.2 Eventlet Middleware Implementation

We developed a distributed Eventlet middleware following the architecture presented in Section 6.1. Our middleware is implemented in Java and builds on top of a pub/sub system. We provide an adapter layer for pub/sub interactions to enable the integration of different pub/sub middleware; adapters provide the basic pub/sub API (cf. Section 3.1) to the components of the Eventlet middleware. The communication between the distributed components of the middleware, i.e., Eventlet instances and Eventlet monitors, is decoupled and asynchronous; this fosters scalability.

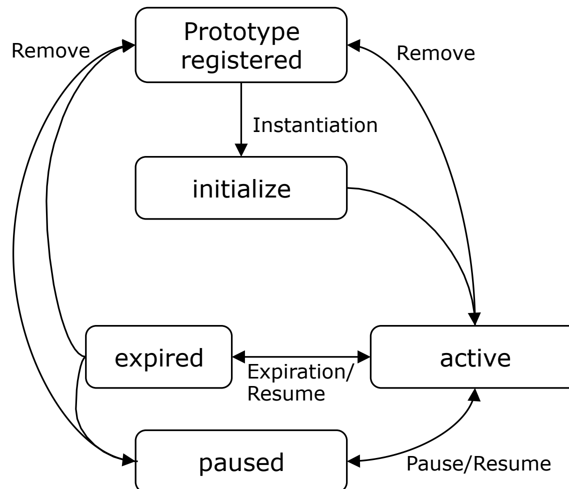


Figure 6.4: Eventlet life cycle states: After instantiation an active Eventlet instances can expire or it can be paused or removed.

JMS Pub/Sub Adapter

We implemented an adapter to use Java Message Service (JMS) brokers as event bus. JMS is the de-facto industry standard for asynchronous messaging; messages are exchanged via a JMS broker network, e.g., IBM WebSphere MQ or Apache ActiveMQ (cf. Section 2.1.2). Our JMS adapter handles the connection setup and provides an abstraction for event types. Our adapter layer supports att/val pairs as well as XML as representation for events. Figure 6.5 shows a temperature event in both representations. Att/val events as well as XML events are mapped to corresponding JMS objects inside the adapter. XML events are represented as JMS text messages where the message body contains the XML source. Att/val events are represented with JMS message properties, i.e., the JMS header is used to store the data. Att/val events can be used with arbitrary JMS brokers; XML events require support for subscriptions specified on the XML content.

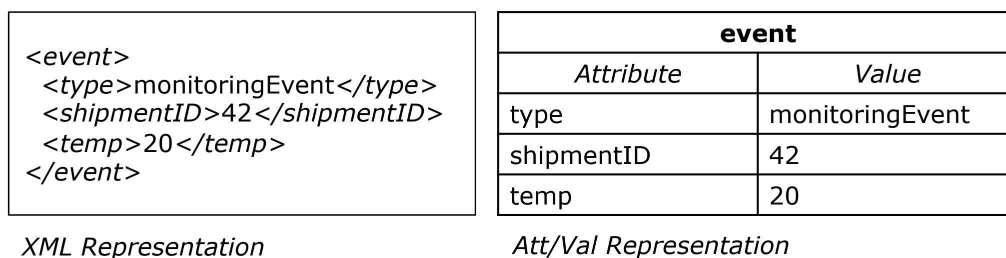


Figure 6.5: Event representation

All JMS brokers support topic-based pub/sub with filtering on message header attribute values via so called message selectors. In addition, some brokers support XML events, e.g., Apache ActiveMQ. The syntax of subscription filters – in form of JMS message selectors – differs for att/val events and XML events. For att/val events the subscription filters are specified as standard JMS message selectors in a SQL-like syntax; for XML the filters are specified in XPath. This results in different formats for instantiation and constant expressions depending on the underlying infrastructure. With

JMS brokers that support both event representations, Eventlet prototypes defined on att/val filter expressions or XPath filter expressions can exist in parallel. However, event producers have to take care about the format of published events. For events to be usable by att/val Eventlets as well as by XML Eventlets, published events have to contain both: the att/val representation (in the event header) and the XML representation (in the event body). In our implementation we use a single topic as event bus; all event producers publish events to this topic, all consumers (Eventlet monitors and Eventlet instances) subscribe to this topic using dynamically derived message selectors.

We trade more elaborated content-based subscription capabilities of alternative pub/sub systems, e.g., filter subsumption, for the use of JMS. JMS has the advantage of being a standardized industry-strength API that allows the use of well-tested JMS brokers. JMS is also the technology used in many Enterprise Service Buses (ESBs); Apache ServiceMix, for example, uses ActiveMQ as ESB. Thus, the use of JMS enables the reuse of already deployed enterprise application technology by our Eventlet middleware.

Middleware Inter-Component Communication

The Eventlet middleware is designed for scalability as Eventlet monitors and Eventlet instances can run distributed across multiple machines. To achieve this distribution, queue- and topic-based inter-component communication is integrated in the middleware. Just like the event bus transports events from producers to consumers, the Eventlet middleware uses a command bus for asynchronous and decoupled communication between the components. This command bus is a set of JMS queues and topics to which the different components subscribe using message selectors. To start Eventlet monitors and Eventlet prototype instances on different servers, the Eventlet Monitor Server and Eventlet Instance Server are started on each machine participating in the middleware system. These servers form the Eventlet manager and connect to the command bus. In the current implementation, queues deliver commands round-robin to all connected servers. Two queues exist, one to trigger the creation of Eventlet monitors, a second to trigger the creation of Eventlet instances. The distributed architecture is shown in Figure 6.6. With our command dissemination infrastructure, components can be addressed at different granularity levels. It is possible to send commands to:

- a specific Eventlet instance;
- all Eventlet instances, monitors, and instance servers;
- all Eventlet monitors;
- all active Eventlet instances, not only to instances of a certain Eventlet prototype;
- a specific Eventlet monitor; and
- all Eventlet instances of a prototype.

The creation of Eventlet monitor instances and Eventlet instances is triggered via commands sent to dispatch queues to which Eventlet monitor servers and Eventlet instance servers subscribe:

- *Eventlet monitor dispatch command:* This command is sent at the registration of an Eventlet prototype. After the prototype is stored in the repository, an arbitrary Eventlet monitor server can react to this command and create the Eventlet monitor instance.

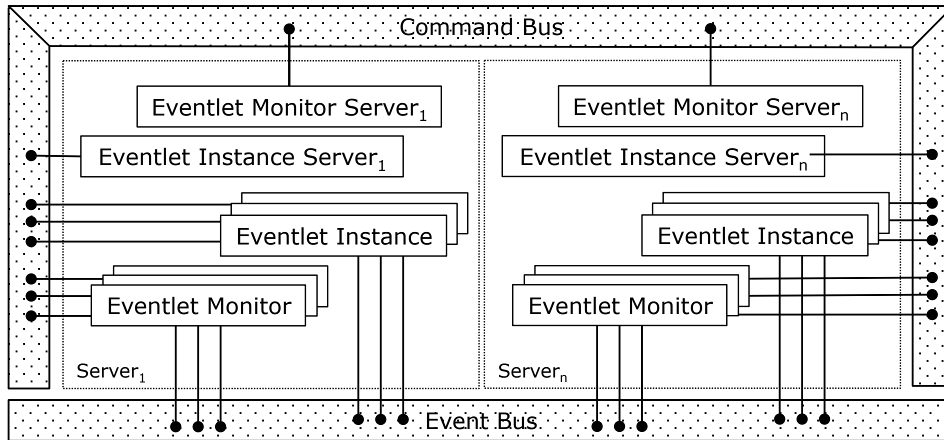


Figure 6.6: The Eventlet middleware is distributed across machines: Eventlet Monitor Servers and Eventlet Instance Servers receive commands and trigger the creation of Eventlet monitors and Eventlet instances.

- *Eventlet dispatch commands:* Eventlet monitors send *Eventlet dispatch commands* to trigger the creation of Eventlet instances. The commands are received by Eventlet instance servers, which then access the repository to create Eventlet instances that may run on different nodes.

The middleware also supports three different Eventlet management commands to control the lifecycle of Eventlet instances (cf. Figure 6.4) and Eventlet monitors:

- *Pause:* Pause commands pause Eventlet instances; instances remain active and keep state but do not receive events anymore and the validity check is disabled. Although Eventlet instances keep state, this state becomes obsolete as events are missed; the handling of potentially occurring inconsistencies depends on the use case. Pause commands sent to an Eventlet monitor prevent the creation of new Eventlet instances. The state of already created Eventlet instances remains unchanged.
- *Resume:* Resume commands reactivate paused Eventlet instances and monitors.
- *Remove:* Remove commands sent to Eventlet instances trigger the `onRemove` method call and stop the instance. Depending on the configuration, removed Eventlet instances are recreated or not when the Eventlet monitor is still active and matching events arrive again. Removing an Eventlet monitor causes all associated Eventlet instances to be removed as well.

Eventlet Prototype Implementation

In our middleware Eventlet prototypes are represented by Java classes. An Eventlet prototype inherits its functionality from its superclass; the superclass contains methods to evaluate the constant and instantiation expressions as well as code to connect the Eventlet instances to the pub/sub system. The superclass further implements a message listener interface to react on incoming messages. All of this is transparent to the Eventlet developer who only needs to implement the four core methods and to provide the appropriate constant and instantiation expressions as described in Section 3.2.3.

Listing 6.1 shows a sample Eventlet prototype for the monitoring of shipments. It reacts on att/val events of type *monitoringEvent* (cf. Listing 6.1, Line 3), as shown in Figure 6.5. A new Eventlet instance is created for each shipment (cf. Listing 6.1, Line 4). Upon instantiation, a shipment-depending temperature threshold is retrieved and used to check for temperature violations. The Eventlet prototype also specifies a timeout (cf. Listing 6.1, Line 5). The code in Listing 6.1 is a fully functional Eventlet prototype and no additional code is needed to allow its registration with the Eventlet middleware. However, more application logic has to be added to implement the threshold lookup upon instantiation. Since an Eventlet prototype is a Java class, it can use all Java constructs and integrate external libraries. The implementation of Eventlet prototypes as Java classes allows Eventlet instances to keep state. In this context it is important to note that the `onEvent` method is invoked asynchronously so that synchronization is required, for example to access variables.

Listing 6.1: Sample Eventlet prototype for shipment monitoring with att/val events.

```
1  public class ShipmentMonitoring extends TimeoutEventlet {
2    // Eventlet prototype meta data
3    public static String constantExpression = "(type='monitoringEvent')";
4    public static String instantiationExpression = "shipmentID";
5    public static long validityExpression = 120000;
6
7    // Constructor (auto generated)
8    public ShipmentMonitoring(String eventletName, String instantiationValue)
         {
9      super(eventletName, instantiationValue, constantExpression,
          instantiationExpression, validityExpression); }
10
11   // Eventlet prototype runtime code
12   public int tempThreshold;
13
14   @Override
15   public void onEvent(Event e) {
16         int temp = e.getVal("temperature");
17         if (temp > tempThreshold) raiseAlert();
18         super.onEvent(e);
19   }
20
21   @Override
22   public void onInstantiation() {
23     // Retrieve temperature threshold for shipment
24     tempThreshhold=thresholdLookup(this.getInstantiationValue());
25   }
26
27   @Override
28   public void onExpiration() { }
29   @Override
30   public void onRemove() { }
31 }
```

The Eventlet meta data for an XML variant of the shipment monitoring Eventlet prototype is shown in Listing 6.2. Constant and instantiation expression are XPath expressions; the event data is now retrieved as string (cf. Listing 6.2, Line 9) which contains the XML.

Listing 6.2: Sample Eventlet prototype for shipment monitoring with XML events.

```
public class ShipmentMonitoring extends EventletXml {
  // Eventlet prototype meta data
  public static String constantExpression  =
    "/event/type[. = 'monitoringEvent']";
  public static String instantiationExpression = "/event/shipmentId";

  // Eventlet prototype runtime code
  @Override
  public void onEvent(Event e) {
    String xmlPayload = e.getPayload();
  }
}
```

Eventlet Prototype Instantiation

Upon the registration of a new Eventlet prototype a command is sent to the Eventlet monitor dispatch queue. The Eventlet monitor server retrieving this command creates the Eventlet monitor. To allow all Eventlet monitor code to be generic, we use Java Reflection to identify Eventlet prototype classes and create new instances. Adding a new Eventlet prototype thus does not require code modifications; the Eventlet is identified by a string, which is used to find the respective class and constructor. The command to create an Eventlet monitor for the shipment monitoring Eventlet prototype is shown in Listing 6.3:

Listing 6.3: Monitoring Eventlet registration with implicit instantiation.

```
commandConnection.sendToEventletMonitorDispatchQueue(
  new EventletMonitorDispatchCommand(
    "de.tudarmstadt.dvs.eventlets.repository.ShipmentMonitoring"));
```

To create Eventlet instances, either implicitly or explicitly, a command is sent to the Eventlet dispatch queue. This command contains the Eventlet name as well as the concrete instantiation value. For the creation of a shipment monitoring Eventlet for Shipment No. 42 the command is shown in Listing 6.4:

Listing 6.4: Explicit instantiation of monitoring Eventlet.

```
commandConnection.sendToEventletDispatchQueue(
  new EventletDispatchCommand(
    "de.tudarmstadt.dvs.eventlets.repository.ShipmentMonitoring", "42"));
```

The Eventlet instance server receiving this command then instantiates the corresponding Eventlet. Active Eventlet instance servers receive dispatch commands round-robin. A physical machine can

start multiple Eventlet instance servers to implement load balancing. To improve locality of components the round-robin distribution can be replaced by other mechanisms so that routing algorithms of an underlying broker network can work more efficiently.

A newly created Eventlet instance knows which value triggered its instantiation; thus, it can derive its subscription for events associated with an entity instance. Depending on the type of validity condition, a parallel task is started with the creation of an Eventlet instance. In case the validity condition specifies a timeout, this is periodically checked. The `onExpiration` method is invoked when the validity expression is violated.

6.2.1 Eventlet Middleware Interfaces

As shown in Figure 6.7 our Eventlet middleware provides different interfaces for registering and controlling Eventlets. With the native client, the registration of Eventlet prototypes and the explicit instantiation of Eventlets is directly integrated into the middleware. The middleware is responsible for the construction of command objects and for establishing a connection with the command bus. The JMS interface gives external applications the possibility to send commands directly to the Eventlet middleware command bus. In this case, the connection to the command bus has to be established manually. Further, Eventlet middleware libraries have to be included into the application class path so that the required command objects can be created. For the integration of event stream processing with business processes we developed an Eventlet middleware interface to support Event Stream Processing Task (ESPT) execution. The implementation of ESPTs in business processes can be realized with Eventlets; a shipment monitoring ESPT, for example, corresponds to a shipment monitoring Eventlet. The semantics of ESPT execution (cf. Section 5.2.2) are supported by the Eventlet middleware. The `EsptInstantiate` and `EsptRegister` invocations provide the Eventlet middleware with the metadata to explicitly or implicitly create Eventlet instances. The Eventlet prototype associated with an ESPT is identified via the `EsptName`. The Sub Stream Attribute is the name of an event attribute, e.g., shipmentID. For XML events, Event Stream Filter and Sub Stream Attribute are specified as XPath expressions on the event content. We added a web service interface to the Eventlet Manager to support the automated execution of ESPTs by process execution engines; the interface accepts service invocations as described in Section 5.2.2 and uses the internal command bus to start or stop Eventlet Monitors and Eventlet instances. The web service interface is implemented as Java Enterprise application.

6.3 Privacy Concept

SPUs process entity-centric streams of events. In this context, privacy can become an important issue. While the monitoring of shipments might not involve the processing of highly sensitive data, an SPU for patient monitoring in hospitals (cf. Section 3.2) certainly deals with confidential event streams. Thus, the data transport from event sources to event consumers, i.e., to SPUs, must be confidential. SPUs rely on a pub/sub system; our Eventlet middleware, for example, uses JMS. However, the principle of pub/sub systems contradicts the demand for privacy. Event producers publish events and brokers match events with subscriptions following a one-to-many communication scheme. This decoupled information exchange requires brokers to have access to the event data to be able to perform the matching with subscriptions [174].

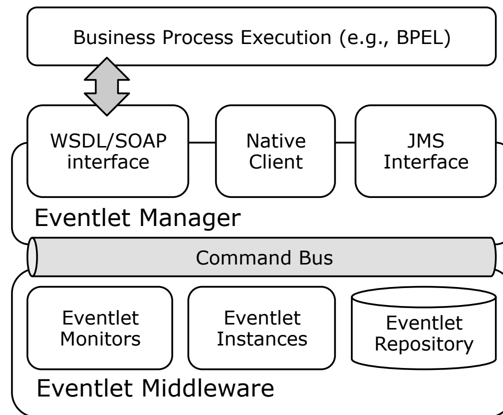


Figure 6.7: The Eventlet middleware provided a web service interface for the integration with business process execution environments.

While at a small scale a trusted broker network can be a feasible solution to ensure privacy, trusted large-scale broker networks are hard to implement. Broker networks may encompass brokers operated by different companies and at different locations without a central control authority. Thus, approaches have been developed that allow privacy-preserving pub/sub communication with an untrusted broker network [123, 160]. In these systems brokers can route events without access to the event content.

In [123], the Paillier homomorphic encryption scheme [135] is used to enable the evaluation of blinded subscriptions against blinded event attributes. This enables routing of events through a network of untrusted brokers and ensures confidentiality of subscriptions and events. However, each subscriber needs to contact a Trusted Third Party (TTP) for the subscription blinding process. In [120, 121] we enhance the approach presented in [123]. We improve performance and adapt the encryption scheme to minimize the contact with a TTP. This allows the application of our approach in domains with high subscription dynamics, i.e., frequently changing subscriptions or frequent new subscriptions. Our Eventlet middleware is such an application domain: the creation of each new Eventlet instance involves issuing a subscription. Especially in use cases where many Eventlet instances are active and the instances have short life cycles, a TTP contact upon each subscription is not feasible anymore.

In the following we apply our privacy-preserving pub/sub approach to the Eventlet middleware. We implemented our privacy-preserving pub/sub solution in the Apache ActiveMQ JMS broker. Since our Eventlet middleware is implemented on top of JMS, the privacy-enhanced broker integration does not require the adaption of interfaces.

Privacy-preserving Publish/Subscribe

Figure 6.8 shows our privacy-preserving pub/sub scheme. It relies on the homomorphic properties of the Paillier encryption, i.e., computations performed on encrypted data correspond with computations on unencrypted data. In our context, these homomorphic properties allow the comparison of encrypted values whereas the comparison result also applies to the unencrypted data. Our scheme distinguishes between privacy-preserving routing of events, i.e., matching subscriptions with events,

and a payload encryption. To perform routing decisions, a decryption of events is not required: only the event consumers require access to the actual unencrypted data. For the matching of events and subscriptions, homomorphic properties of the encrypted data are sufficient. A homomorphic encryption scheme allows brokers to compare encrypted values and to make correct routing decisions without having knowledge about the unencrypted data. This observation is the foundation for our privacy-preserving pub/sub scheme with minimal TTP contact. We use a so called blinding operation: blinding is a one-way randomized encryption that preserves homomorphic properties. Brokers can compare blinded values and make routing decisions depending on the comparison result; our encryption scheme ensures, that the comparison result also holds for the unencrypted data. The blinding operation is also randomizing, i.e., blinding the same value twice leads to different blinded values whereas the homomorphic properties still hold. This prevents brokers from learning from the distribution of encrypted values. Our trust model is as follows: we assume brokers to be honest but curious, i.e., brokers follow the pub/sub protocol, but are not trusted for confidentiality of publications and subscriptions. They also may collude with subscribers. We assume that the TTP is fully trusted. We assume that publishers keep the secrets they receive from a TTP confidential and follow blinding protocol; periodical re-authentication with a TTP can be used to ensure privacy in case of malicious publishers. Subscribers are not trusted in our scheme.

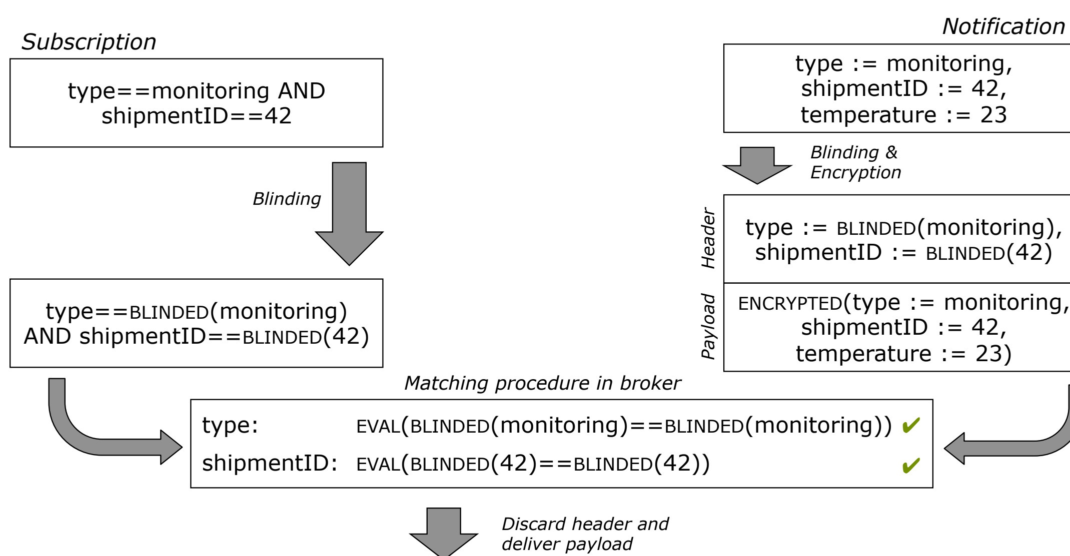


Figure 6.8: The privacy-preserving pub/sub scheme requires blinding of subscriptions and corresponding event attributes; further, a separate payload encryption is applied.

Prior to privacy-preserving pub/sub operations, publishers, subscribers, and brokers contact a TTP to retrieve parameters required for the blinding and encryption as well as for the comparison and decryption operations. As shown in Figure 6.8, a subscriber constructs a filter expression (subscription) and then blinds the attribute values. This subscription is then used to register with a broker. On the event publisher side, an initial TTP contact is also prerequisite for the subsequent publishing of events. The next step is blinding as well as encrypting the event data. The portion of the event data that can be part of subscription expressions needs to be blinded. In a content-based pub/sub system with att/val events this can involve all event attributes. Blinding is a one-way operation and subscribers cannot gain access to the plain event data from blinded values. Thus, a separate payload

encryption has to be applied. In our system, an attribute-based group key management scheme, as presented in [122], is used.

The event with a blinded and an encrypted portion is sent to the broker where the matching between registered subscriptions and the event content is performed. The comparison between blinded values requires a multiplication in modulo arithmetic with the parameters received from the TTP. The result of this multiplication tells whether there is a *greater or equal than* or a *less than* relation between two values. An equality check is not supported directly; rather, subscribers have to split an equality check in two comparisons. The equality check `value == 42`, for example, is split in `(value >= 42) AND (value < (42+1))` by the subscriber. On the one side, this introduces computational overhead, on the other side, malicious brokers cannot distinguish between range and equality filter expressions anymore. Depending on the outcome of the comparison inside the broker, the broker can perform routing operations and forward events to interested subscribers.

Besides a computational overhead, our privacy-preserving pub/sub scheme also influences scalability properties of pub/sub systems. The whole set of filter subsumption mechanisms cannot be applied anymore: two subscriptions `(value > 5)` and `(value > 7)` cannot be subsumed to `(value > 7)` and pushed close to the event source since brokers are not able to see values in plain text. Still working, however, is splitting of subscriptions. The subscription `(value1 > 5) AND (value2 > 7)`, for example, can be split and the first part `(value1 > 5)` can be moved to a broker close to the event source. This does not reduce the overall computational effort, but reduces network utilization since events are filtered early.

The blinding operation is applied to attribute values; attribute names used in subscriptions and publications are not affected and remain in plain text. To prevent attackers from learning from attribute names, meaningless attribute names can be used, e.g., `value1`. It is also possible to apply a hash function to the attribute name; however, this allows dictionary attacks to retrieve the attribute name. In either case - with meaningless or with hashed attribute names - subscribers and publishers have to possess shared knowledge about attribute names and attribute semantics; this is intrinsic to the pub/sub approach.

To secure event data and allow decryption at event consumer site, a separate payload encryption scheme has to be applied. This is independent from the blinding mechanism to achieve privacy-preserving pub/sub functionality. In [120, 121] we use an attribute-based group key management scheme (AB-GKM) [122] for payload encryption. AB-GKM is an enhanced broadcast group key management (BGKM) scheme, presented, e.g., in [24, 49]. In BGKM subscribers receive an initial secret from a TTP; during operation, data is broadcast that - in combination with the initial secret - can be decrypted. For all broadcast data, publishers can decide which subset of registered subscribers should be able to decrypt data; no additional TTP involvement is required in this step. AB-GKM extends BGKM with attribute-based access control policies (ACP): a subscriber ACP is a set of attribute conditions that has to be fulfilled in order to gain access to the data. Publishers decide which access control attribute values an event has and, by this, determine which subscribers can decrypt the data. A possible ACP is, for example, `((priority == normal) AND (type == temperatureEvents)) OR (priority == high)`; with this ACP a subscriber can decrypt temperature events with normal priority as well as all high priority events. Publishers specify priority and event type and determine implicitly which subscribers are able to decrypt the event.

Privacy-preserving Pub/Sub Integration in Eventlet Middleware

Figure 6.9 shows the integration of our privacy-preserving pub/sub scheme with our Eventlet middleware. Event publishers act independently of our middleware; they contact a TTP and are then able to blind attribute values and to encrypt payload. At subscriber site, two components of our middleware issue subscriptions: Eventlet monitors and Eventlet instances. Eventlet monitors contact the TTP; they are then able to blind subscriptions. Eventlet monitors for implicit instantiation issue subscriptions built from the constant expression. Subsequently, Eventlet instances issue subscriptions built from the constant expression as well as from an instantiation value set (see Section 6.1.2).

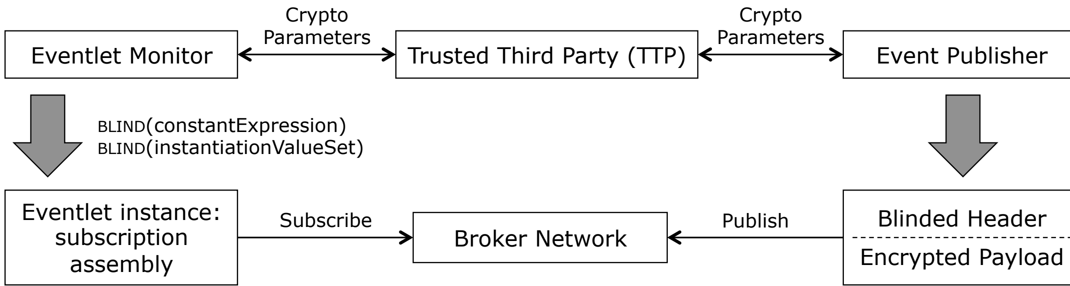


Figure 6.9: Integration of privacy-preserving pub/sub with the Eventlet middleware.

We identified three alternative approaches for the blinding of the subscription values of Eventlet instances:

1. Eventlet monitors share cryptographic parameters with Eventlet instances;
2. Eventlet monitors perform blinding of subscriptions themselves and forward blinded values at Eventlet instance creation; and
3. Upon creation, Eventlet instances contact a TTP to receive the secrets required for the blinding operation.

When the cryptographic parameters are shared by an Eventlet Monitor, Eventlet instances can blind, and thus issue, arbitrary subscriptions. When subscriptions are blinded by the Eventlet monitor, it acts as trusted authority and control component for all Eventlet instances. In this case Eventlet instances cannot issue arbitrary subscriptions on their own, they rely on blinding operations performed by the Eventlet monitor. Shifting the blinding operations from the Eventlet monitor to the Eventlet instances removes load from Eventlet monitors and increases scalability. At the same time, Eventlet instances have to be fully trusted since they can issue arbitrary subscriptions. When Eventlet instances have to contact a TTP before issuing a subscription, the TTP has to handle many requests. Especially in cases with high Eventlet instantiation dynamics, this is not suitable anymore. This bottleneck initially motivated the development of our encryption scheme where arbitrary subscriptions can be blinded after an initial TTP contact.

As the blinding of subscriptions, the decryption of the event payload by Eventlet monitors and Eventlet instances requires secrets obtained from a TTP. Best privacy protection is achieved by an explicit TTP contact of each Eventlet instance where the Eventlet instance specific subscription is used as access policy, e.g., `(type == monitoringEvent) AND (shipmentID == 42)`. This guarantees that no other Eventlet instance is able to decrypt data. An Eventlet monitor for implicit subscription,

however, needs to be able to decrypt all event data that matches the constant expression and the instantiation expression since it has to trigger the creation of Eventlet instances when demanded. An additional level of security can be added by an Eventlet monitor specific event payload encryption. Event publishers then perform a blinding of attributes for the subscription matching, they encrypt attributes identified via the instantiation expression by Eventlet monitors, and they encrypt the complete event for the final recipients. Due to the pub/sub scheme, events arriving at Eventlet monitors fulfill the constant expression. An Eventlet monitor is then able to decrypt only the instantiation values to decide upon the creation of new Eventlet instances. Eventlet instances register with their instantiation value set with a TTP and are subsequently able to decrypt the payload of events associated with this instantiation value set, e.g., events for Shipment No. 42.

As for the blinding operation, a TTP contact of each Eventlet instance is often not feasible. Alternatively, secrets can be shared between an Eventlet monitor and its associated Eventlet instances. This allows Eventlet instances the decryption of all events that match the constant expression of the Eventlet monitor. This becomes a security issue when attackers manage to inject malicious Eventlet instances: these instances may subscribe to all events and will then be able to decrypt all events related to an Eventlet monitor, e.g., a malicious Eventlet instance can decrypt monitoring events for all shipments rather than monitoring events for a single shipment only.

The above presented realization alternatives of our privacy-preserving pub/sub scheme show the tradeoff between performance & scalability and the level of privacy & security. For best privacy and security, each component instance of our middleware retrieves its own secrets from a TTP; in highly dynamic scenarios, however, the TTP becomes a bottleneck. For best performance and scalability Eventlet monitors share secrets with all Eventlet instances. In either case, however, the privacy of event data against the brokers is assured since brokers are not able learn anything about the events they are routing.

In general, adding privacy-preserving techniques to a pub/sub system introduces an overhead due to the cryptographic operations. This decreases throughput and increases computational effort. To quantify the overhead, we present a performance evaluation of our privacy-preserving pub/sub scheme in Section 7.2.

6.4 Transformation Concept

The subscriptions issued by Eventlet monitors and Eventlet instances are specified to match content of events. This, however, requires knowledge about the structure and semantics of events. For att/val events, the attribute names as well as the semantics of attribute values need to be known, i.e., subscribers and publishers have to share knowledge. On the one hand, event-based interaction has the advantage of anonymous one-to-many communication (publishers do not know about consumers); on the other hand, consumers require knowledge about event structure and semantics. When events are consumed by multiple subscribers – a desirable behavior in Event-based Systems (EBSs) – establishing this common knowledge base is cumbersome: with n producers and m consumers n times m knowledge exchanges are necessary. To address this heterogeneity effect in EBSs, we introduced a middleware-based transformation approach for events [65, 73, 74]. We use the term *context* to refer to a common understanding of event structure and semantics between producers and consumers. Event producers and consumers that share the same context are able to understand each other. An

example for context is the unit system used for attribute values in events. Some publishers might use the imperial system while others use a metric system. With our approach it is possible to mediate between these different contexts, i.e., it allows events produced in one context to be consumed by subscribers that reside in a different context. We achieve this by introducing rules that specify the transformation of events from one context to another, e.g., the conversion from imperial system to metric system. Our approach is generic and can be applied to arbitrary pub/sub systems; our reference implementation is JMS-based and realized with Apache ActiveMQ. In the following we illustrate how our transformation approach can be applied to Eventlets.

In our Eventlet middleware, Eventlet monitors as well as Eventlet instances issue subscriptions based upon constant and instantiation expressions. The constant and instantiation expressions are specified by developers as part of the Eventlet metadata (associated with Eventlet prototypes). The constant and instantiation expressions require knowledge about event structure and semantics. Developers need to know the attribute names as well as meaning of attribute values in order to write Eventlet prototypes. In our shipment monitoring example, developers need to know that there is an attribute named *shipmentID* that holds a unique shipment identifier. Developers also need to know the name of the attribute that holds the reported temperature as well as the unit system the temperature is reported in. Given this scenario, the following two cases illustrate the demand for a mediation layer that transforms events:

- A shipment might be transported by different vehicles through various regions. Most likely, different hardware is deployed in different vehicles, e.g., temperature sensors of different vendors where temperature events have different formats or even report temperature in different unit systems, e.g., in Celsius or in Fahrenheit. An Eventlet instance that monitors a certain shipment requires a unified view on these different types of events, i.e., all events that arrive at an Eventlet instance must use unified attribute names and follow the same attribute value semantics.
- Different Eventlet prototypes might also reside in different contexts. One Eventlet prototype might require temperature events in Fahrenheit while another Eventlet prototype might require temperature events in Celsius. Both Eventlet prototypes, however, should consume events published by the same producers and benefit from the one-to-many interaction principle of pub/sub.

Our transformation approach provides the flexibility to fulfill such demands. As shown in Figure 6.10 we add a layer to the pub/sub middleware, which performs event transformations and allows seamless communication between participants that reside in different contexts. Each participant in this setting specifies a set of transformation rules that describe the transformation from one context into another context. In theory, this does not necessarily reduce the complexity of event transformations, i.e., when each participant specifies a transformation from its context to all other contexts, the complexity remains at $n \times m$. To reduce this complexity, we introduce the concept of a *root context*. The root context is a specific event type with known structure and semantics. At least one participant should reside in this root context to avoid unnecessary transformations. All other participants specify transformation rules with respect to this root context, i.e., how events have to be transformed from the participants' context to the root context. Our system also supports multiple root contexts, i.e., for one event category (e.g., temperature measurements) the root context is defined by one

participant, for another event category (e.g., position events) the root context is defined by another participant.

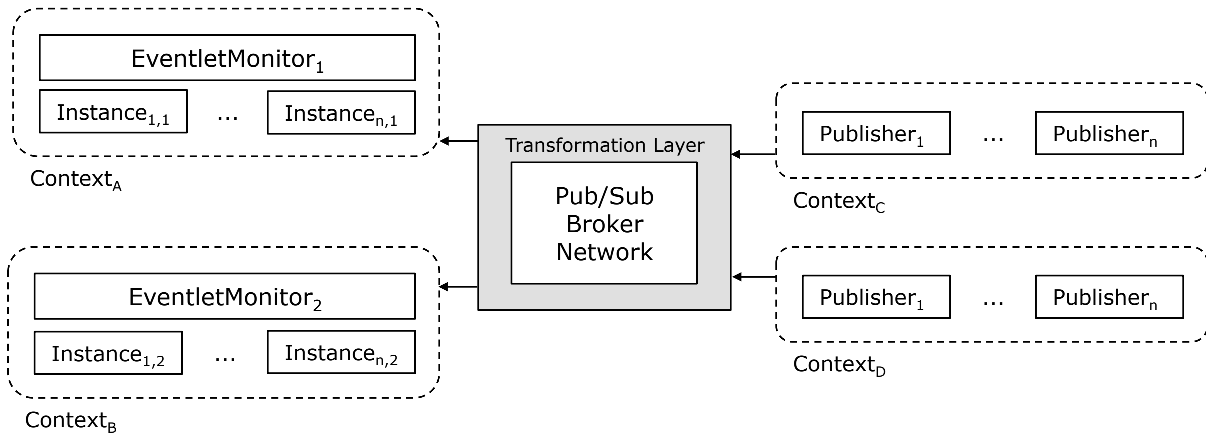


Figure 6.10: Integration of context transformation with Eventlets: Eventlet monitors and their associated Eventlet instances can reside in different contexts.

Listing 6.5 shows a sample context definition for our shipment monitoring example. When temperature measurement events are, for example, published within the United States (us) context, Eventlet monitors and Eventlet instances residing in a European (eu) context need to specify a transformation. Our context definition consists of three blocks: type definitions, mappings, and transformation rules. The type definitions are the basis for the application of transformation rules; type definitions define the structure of events and specify the events to which the context transformation should be applied. Mappings are necessary to define the event type that results from the application of the transformation; mappings also define a direction: transformations are applied to incoming or outgoing events. Finally, the rules themselves specify the transformation in detail; transformation rules refer to functions that perform the transformation, e.g., `toCelsius`. The `toIdentity` function explicitly states that this part of an event should not be transformed; this can be necessary in case transformations for the super type of an event, e.g., `com.logistics.eu`, are part of the context definition. The expression `MonitoringEvent.shipmentData` refers to a subtype (set of attributes) that holds, e.g., the shipmentId. At runtime, incoming monitoring events are transformed according to the specified context definition. Next, subscriptions are evaluated against the transformed event, i.e., subscriptions refer to attribute names and attribute value semantics of the subscriber's contexts. In case of a match, the event is forwarded to the next broker node or the final recipient.

Listing 6.5: Context definition.

```
1  <types>
2    com.logistics.eu.MonitoringEvent
3    com.logistics.eu.MonitoringEvent.shipmentData
4  </types>
5  <mappings>
6    <mapping from="com.logistics.us.MonitoringEvent"
7    to="com.logistics.eu.MonitoringEvent" direction="outgoing" />
8  </mappings>
9  <rules>
10   <rule pattern="MonitoringEvent.temperature" function="toCelsius" />
```

```
11  	<rule pattern="MonitoringEvent.shipmentData" function="toIdentity" />
12  </rules>
```

Our transformation approach supports transformation rules with different nesting levels (cf. lines 10 and 11 in Listing 6.5). Patterns can also reference complete (sub)types (Listing 6.5, line 11) or specific attributes (Listing 6.5, line 10). In [65, 74] we discuss the resulting conflicts at transformation rule resolution and introduce priorities for rule application based upon precedence. We also present the formal language model underlying our transformation approach. Our transformation approach can be integrated with the Eventlet middleware: Context definitions can be specified as part of the Eventlet metadata. Upon the registration of an Eventlet prototype with the Eventlet middleware, the context definition is deployed. During the deployment it is checked whether the necessary functions, e.g., `toCelsius`, are supported by the transformation layer of the pub/sub system. If the desired function is not yet available, this is reported to the developer for further consideration.

We implemented our transformation approach as plugin in Apache ActiveMQ. Applications, like our Eventlet middleware, that are based upon JMS, can be adapted easily to support transformations; we extended the JMS API with methods to register context definitions and to set the context. These methods have to be invoked at JMS connection setup. Our implementation generates code from context definitions; this code is applied to events by the middleware. Since in EBSs event producers and consumers can dynamically join and leave the system our transformation approach allows dynamic modification of contexts, i.e., when a new producer joins the system it can specify its context based upon already registered context definitions. Further, the performance of the pub/sub system is crucial in EBS. Our middleware-based transformation approach adds an overhead to the middleware rather than to publishers and subscribers. This is the downside of the reduced complexity achieved with a central transformation authority via root contexts. However, our evaluations in [73, 74] show that the performance loss is acceptable while the reduction of complexity is significant.

6.5 Summary

In this chapter we introduce Eventlets – our implementation of the SPU container model. Eventlets are executed and managed by a distributed Eventlet middleware. The implicit instantiation of Eventlets is triggered by Eventlet monitors. Eventlet instances dynamically issue entity instance centric subscriptions, e.g., for events that belong to a certain shipment. Eventlet instances follow a lifecycle and run as autonomous units. The implementation of our Eventlet middleware is Java-based. We rely on JMS for the pub/sub functionality; JMS is an industry-strength messaging standard often used in enterprise applications. Thus, our Eventlet middleware integrates well with existing enterprise application landscapes.

In addition to our Eventlet middleware architecture and implementation, we discuss heterogeneity and privacy issues of pub/sub systems in the context of Eventlets. Since events are disseminated via a broker network, the protection of privacy is challenging. We present the application of our privacy-preserving pub/sub scheme to address the privacy demands in pub/sub communication. Further, heterogeneity is an issue in pub/sub systems; event publishers and subscribers have to agree on common semantics to be able to exchange events. We address this by applying an event transfor-

mation approach to Eventlets; Eventlet monitors and Eventlet instances can specify transformations that convert events between different structural representations and semantic contexts.

7 Evaluation

Eventlets, as presented in Chapter 6, are an implementation of our Event Stream Processing Unit (SPU) container model that allows the encapsulation of event stream processing application logic. In this chapter we present an evaluation of Eventlets. In Section 7.1, we show performance and software engineering benefits due to Eventlets. We evaluate aspects of our Eventlet approach and show that the use of Eventlets is beneficial from the software development perspective; Eventlets provide automatisms to build scalable and distributed event stream processing applications by introducing an abstraction layer. We show that the influence of this additional layer is small while development of event stream processing applications is significantly simplified. In Section 7.2 we evaluate aspects of our privacy-preserving pub/sub scheme introduced in Section 6.3. Since our Eventlet middleware relies on a pub/sub infrastructure, we analyze the influence of our encryption scheme on pub/sub performance.

7.1 Scalability Benefits with Eventlets

Our concept of SPUs – and its implementation with Eventlets – encapsulates event stream processing application logic in manageable units. The Eventlet middleware provides automatism for deployment and management of these units. This entity-type-centric encapsulation is beneficial for system scalability and system development. The management and deployment automatisms (implicit instantiation and completion of Eventlet instances) simplify the development of event-based applications. The component-based middleware architecture allows the distribution of Eventlet instances and Eventlet monitors. In this section we quantify the scalability benefits of Eventlets as well as the advantages from a software engineering perspective.

7.1.1 Setup and Scenarios

We evaluated our system in a Complex Event Processing (CEP) setup. We used the Esper CEP engine and show the benefits of Eventlets when used in combination with Esper. For reference, we also implement a CEP solution purely based upon Eventlets (without Esper). For comparison with another component-based software model, we also implemented our solution with Java Enterprise on the basis of Java Beans.

In our evaluation we show:

- The scalability limitations of traditional CEP applications and that the distribution provided by Eventlets is necessary for scalable event stream processing;
- That the overhead introduced by the Eventlet middleware compared to a traditional CEP solution is small; and
- That the programming model of Eventlets simplifies the development of distributed event stream processing components.

For our evaluation we used Apache ActiveMQ 5.5.1 as Java Message Service (JMS) broker. We used a distributed test environment with three machines that were connected via a 1 GBit network as shown in Figure 7.1. One machine was used to generate the workload, i.e., to send events to the JMS broker. To reproduce a realistic distributed setup, ActiveMQ ran on a dedicated machine. The third machine was used to execute the actual event processing tasks; it was host for the Eventlet middleware, for the Esper CEP engine, as well as for the Java application server.

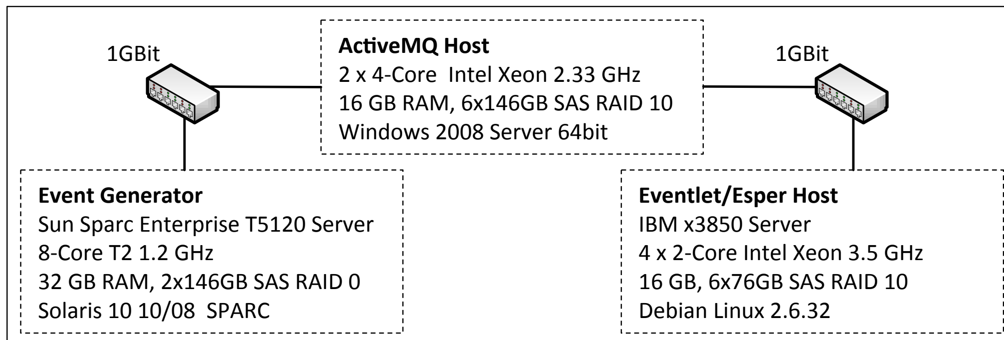


Figure 7.1: Test environment.

We implemented a typical event stream processing application where event processing components receive events via a broker. Events of multiple types and sources are sent to the event bus, which is realized as a JMS topic. Depending on the event type and a threshold defined on the value of one event attribute, a counter variable is increased. We evaluated scenarios with events represented in XML as well as with events consisting of att/val pairs. The XML schema for the XML events is shown in Listing 7.1. Each event contains a type identifier (`type`), a source identifier (`id`), and a random value (`value`). For XML events, we added the type identifier as well as the source identifier as message header attributes so that standard JMS message selectors could be used; the XML code was added as body to a JMS text message. This avoided that the message broker became the bottleneck early: evaluating standard JMS message selectors is less complex than evaluating XPath selectors for events.

Listing 7.1: Structure of events used in Evaluation.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3    <xsd:element name="event">
4      <xsd:complexType>
5        <xsd:sequence>
6          <xsd:element name="type" type="xsd:string"/>
7          <xsd:element name="id" type="xsd:integer"/>
8          <xsd:element name="value" type="xsd:double"/>
9        </xsd:sequence>
10     </xsd:complexType>
11   </xsd:element>
12 </xsd:schema>
```

In our evaluation event producers generate two different types of events and emulate 20 different sources. The event stream processing task is counting the number of events per source where the

value is below/above a type-depending threshold. This comparatively simple task allowed the systematic evaluation of architectural aspects of our Eventlet approach. Rather than evaluating the event processing as such, we focus on scalability and distribution aspects of our container model. A lightweight event processing part ensures that computational resources become saturated due to architectural scalability effects rather than by event processing application logic. This allowed the identification of bottlenecks depending on the system design. We implemented the test case application in four ways:

1. Purely with Eventlets,
2. via CEP queries with Esper (Scenarios Esper A to E),
3. with Eventlets that use Esper (Scenario Eventlet-Esper), and
4. with Java Enterprise Message-driven Beans (MDBs) that use Esper (Scenario Esper-Beans).

The pure Eventlets implementation uses the source identifier as instantiation expression; each Eventlet instance only receives events with a certain source value and counts events corresponding to given thresholds. The comparability of this variant with the Esper variants is limited since the event processing is implemented manually rather than via a CEP engine. All other implementations use the Esper CEP engine to perform the chosen event processing task.

Esper Scenarios A to E

Our implementation on the basis of Esper requires two CEP queries (one for each event type) that count events with respect to a given threshold. The query output is grouped by source identifier. The Esper scenarios A to E follow the common implementation approach for stand-alone CEP applications: Producers send events to a messaging middleware; a CEP engine acts as consumer and subscribes for events. In scenarios Esper A to C a single Esper instance was used and one query per type was registered; the query for one type is shown in Listing 7.2.

Listing 7.2: CEP query used in our evaluation.

```
1 SELECT id, count(*) FROM EvalEvent WHERE (type='TypeA' AND value>0.5) GROUP
      BY id OUTPUT ALL every 10 seconds
```

In Esper scenarios D, E, Eventlet-Esper, and Esper Beans one query per source ID was used, i.e., a total of 20 Esper instances were active with two queries registered per instance. In the Esper-Beans implementation we encapsulate the registration of CEP queries with Esper in message-driven Java Enterprise beans. Such an implementation is suitable for integration in enterprise environments where easy component deployment and lifecycle management is important. The most interesting implementation within the scope of this work is the Eventlet-Esper scenario where we utilize the distribution and deployment capabilities of Eventlets to implement a scalable CEP solution with Esper. In contrast, in the Esper scenarios A to E scalability mechanisms were implemented and configured manually. We varied the number of Esper instances and JMS connection primitives to evaluate scalability.

We implemented different JMS connection strategies. These different strategies depict different levels of distribution controlled via the number of JMS connection primitive objects: JMS connections

Scenario	CF	Esper Instances	Listeners per CF
Esper A	1	1	1
Esper B	1	1	20
Esper C	20	1	1
Esper D	1	20	20
Esper E	20	20	1
Eventlet	20	-	1
Eventlet-Esper	20	20	1
Esper-Beans	AS	20	AS

Table 7.1: Evaluation scenarios: Esper and Eventlet scenarios differ in the number of used Esper instances and JMS connection primitives. CF: JMS Connection Factories; Listener: JMS Message Listener; AS: Determined by Application Server.

are provided by connection factory objects. An application that requires a certain amount of JMS connections can request multiple connections from a single connection factory or create multiple connection factories and request fewer connections per connection factory. The same holds for JMS message listeners that handle incoming messages. We varied the number of Esper instances, the number of JMS connection factories, and the number of JMS message listeners per connection factory to cover different ranges of distribution as shown in Table 7.1.

With respect to the number of connection primitives and Esper intances the Eventlet-Esper and Esper E scenario are equal. The Eventlet-Esper implementation, however, benefits from the distribution and subscription automatisms provided by the Eventlet middleware, as we will show with a code analysis. In the Esper-Bean scenario the connection primitives are managed by the ActiveMQ-to-GlassFish resource connector and are not disclosed to developers. In the distributed Esper scenarios C and E subscription filters are used to receive only events relevant for particular query instances, i.e., a query receives only events containing a certain source identifier. This is shown in Listing 7.3.

Listing 7.3: JMS message selector to filter events by source identifier.

```
1 jmsContext.getSession().createConsumer(jmsContext.getDestination(),
2 "( id = '3' ) AND (( type = 'TypeA' ) OR ( type = 'TypeB' ))");
```

Eventlet-Esper Scenario

In the Eventlet-Esper scenario we used Eventlets to realize distributed event processing with Esper. With the creation of a new Eventlet instance an Esper instance is created and the Eventlet instance passes events on to the Esper instance. Parts of the implementation are shown in Listing 7.4: the constant and instantiation expression specify that one Eventlet should be created per source identifier. In the `onInstantiation` method of the Eventlet the CEP engine is set up and CEP queries are registered. Within each active Eventlet instance the `onEvent` method is then responsible to parse the XML event payload and pass it on to the Esper instance.

Listing 7.4: Implementation of event processing with Esper on the basis of Eventlets.

```
public class EsperEventlet extends Eventlet {
 // Eventlet meta data
 public static String constantExpression  =
   "(type = 'TypeA') OR (type = 'TypeB')";
 public static String instantiationExpression = "id";
 // Esper instance
 public EPServiceProvider esper;
 // Setup of CEP engine
 public void onInstantiation() {
   // XML Event Registration with Esper instance
   URL schema = this.getClass().getClassLoader().getResource("event.xsd");
   ...
   esper.addEventType("EvalEvent", testEvent);

   // Registration of CEP queries
   esper.createEPL("select id, count(*) from EvalEvent where (type='TypeA'
      and value > 0.5) output all every 10 seconds");
   ...
  }

 // Pass events to CEP engine
 public void onEvent(Event e) {
   // Parse payload and build XML tree
   InputSource source = new InputSource(new StringReader(e.getPayload()));
   DocumentBuilderFactory builderFactory = DocumentBuilderFactory.
      newInstance();
   // Send it to ESPER
   Document doc = builderFactory.newDocumentBuilder().parse(source);
   engine.getEPRuntime().sendEvent(doc); }
}
```

Esper-Beans Scenarios

In the Esper-Beans scenario one Message-driven Bean (MDB) is created for each event source and deployed to a GlassFish 3.1.2 application server. Each bean creates an Esper instance and forwards received events to it. A JMS message selector is specified statically for each bean; the resulting subscription is issued upon registration of the bean with the application server. Our scenario requires the implementation of 20 MDBs: one bean for each distinct source identifier value as shown in Listing 7.5 for source identifier 3. The demand for state requires a singleton pattern, i.e., the MDB pool size is one. This ensures that events of one source are processed by a single bean instance only. The encapsulation of CEP functionality in a MDB is similar to the encapsulation inside an Eventlet. However, MDBs are not designed for issuing dynamic subscriptions at runtime; the source identifier needs to be hardcoded.

Listing 7.5: Static configuration part of message-driven bean.

```
@MessageDriven(activationConfig = {
  @ActivationConfigProperty(propertyName = "messageSelector",
  propertyValue = "((type = 'TypeA') OR (type = 'TypeB')) AND id = '3'")}
```

7.1.2 Throughput Measurement Results

We use CPU utilization of the Eventlet/Esper host as the performance indicator for our comparison. On the one hand CPU utilization allows quantifying the overhead of Eventlets. On the other hand scalability across multiple cores is an indicator for good overall distribution capabilities. The latency in our scenario is dominated by the network and message broker; both are not changed in the different scenarios so that we concentrate on CPU measurements here. To determine the limits of the different implementations we increased the event rate up to the point where ActiveMQ flow control throttled down the event producers, indicating that the consumers are saturated. The results of our tests are shown in Figure 7.2.

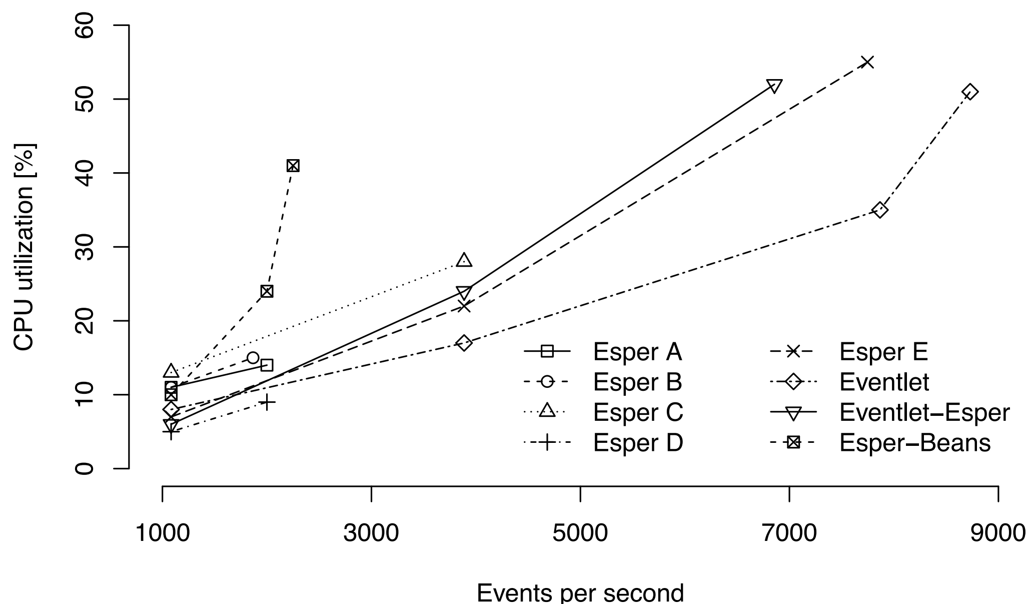


Figure 7.2: CPU utilization of XML event processing realized with Esper and Eventlets.

The Esper scenarios A,B, and D show that a single JMS connection factory is a bottleneck. In these three scenarios the event rate was throttled by the consumer to about 2000 events per second. The CPU is the limiting factor here: since we have eight CPU cores, a utilization of around 10 percent indicates that a single CPU core is saturated, e.g., due to not parallelized methods. This limits event arrival and triggers throttling mechanisms in the broker. In scenario Esper C multiple connection factories are used: a higher CPU utilization and event rate are reached but throttling keeps the event rate at about 4000 events per second making the single CEP instance the bottleneck. In the fully distributed scenario Esper E, an Esper instance for each event source identifier is created and registers for the relevant events using JMS message selectors. This is the implementation nearest to Eventlets

since each Eventlet instance has its own connection factory and subscription. Our evaluation shows that only the distributed setups can process a high volume of events. Even on a single multi-core machine multiple connection primitives are required for scalability across CPU cores.

Our test case can be realized without a CEP engine. The resulting pure Eventlet scenario reaches the highest event rate in our evaluation. Since no external CEP library is included, we gain performance due to reduced complexity. However, often it is not reasonable to abstain from the use of a CEP engine. From a software engineering perspective it is desirable to apply a container model in these cases and encapsulate application logic to foster manageability and scalability. This led to our Eventlet-Esper and Esper-Beans scenarios. The evaluation shows that the Esper-Beans scenario does not perform well compared to the other approaches. It suffers from the complex interplay between application server and message-oriented middleware. The CEP use case does not allow for using large bean pools so that scalability mechanisms of Java Enterprise cannot be applied efficiently. The Eventlet-Esper results show that this scenario is only slightly slower than the dedicated distributed implementation Esper E. This performance loss is introduced by the Eventlet middleware, which adds an additional layer of abstraction to provide the introduced functionality. We think this performance loss is acceptable given the ease of development – which we will quantify in Section 7.1.3 – and the integrated mechanisms for distribution with Eventlets.

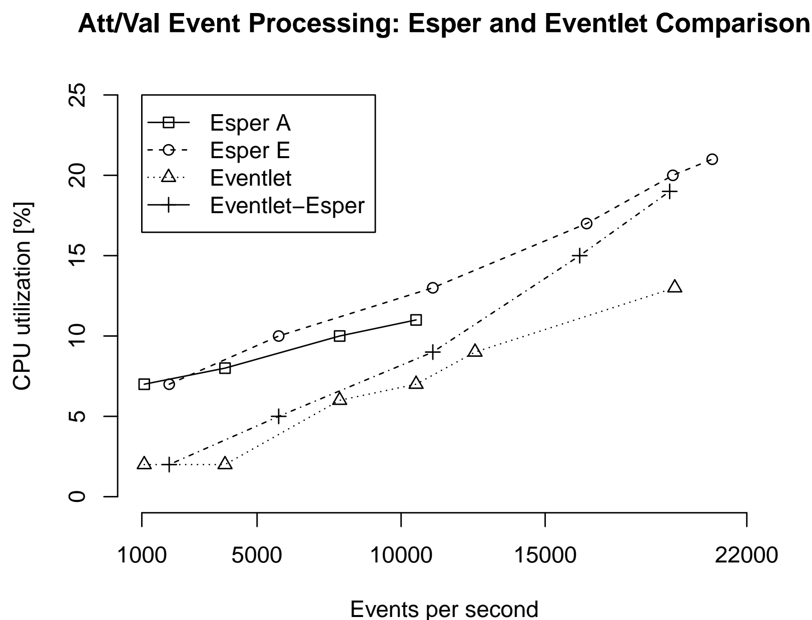


Figure 7.3: CPU utilization of att/val event processing realized with Esper and Eventlets.

In addition to the XML implementation we implemented the basic Esper A scenario as well as the fully distributed Esper E, Eventlet, and Eventlet-Esper evaluation scenarios using att/val events. This avoids computational intensive parsing of XML event payload and leads to higher event rates. The corresponding results are shown in Figure 7.3. As in the XML implementation, the event rate in the centralized Esper scenario A is limited by a single CPU core; since the processing of att/val events is more efficient than XML processing, an event rate of about 10,000 events per second is reached. The distributed scenarios Esper E, Eventlet, and Eventlet-Esper provide better scalability and reach

about 20,000 events per second. At this rate the broker system reached its limit and throttled event producers. As in the XML event processing evaluation, the pure Eventlet implementation has the lowest CPU demand per event. Eventlet-Esper, our implementation with Eventlets that control Esper, has a slightly higher CPU demand. Interestingly, the Esper scenario E has the highest CPU demand. This is also observable in the XML evaluation (Figure 7.2) for a CPU utilizations below 10 percent. This can be caused by the JMS event consumption logic in the pure Esper implementation, which might introduce a higher overhead compared to the event consumption logic in Eventlets. At the same time, however, this event consumption logic enables a slightly higher event rate before the broker is saturated. It also has to be noted that the overall CPU utilization remains below 25 percent in all cases; this shows that our Eventlet approach does not introduce a significant overhead in scenarios with a high rate of att/val events.

7.1.3 Simplified Software Development with Eventlets

The SPU container model, and its implementation in form of Eventlets, introduces an abstraction layer to event stream processing. The purpose of this additional layer is the simplification of the development of distributed event stream processing applications; it provides modularization and a separation of concerns. An additional layer in the software stack introduces an overhead; this loss in performance is traded for software engineering and scalability benefits. Our performance evaluation shows, that the decrease in performance is rather small compared to a dedicatedly developed distributed event stream processing application. This means that Eventlets are well suited for the encapsulation of event stream processing logic in entity-type-centric units from a performance perspective. In this section we show the advantages in terms of software engineering. Eventlets reduces the amount of code programmers have to write to implement distributed event stream processing applications. We analyzed the code of our fully distributed Esper E and Esper-Beans scenario. We compared it with the Eventlet-Esper implementation and counted core application logic code; this excludes code that is automatically generated by modern IDEs, i.e., class headers, exception detection, bean configurations, and constructors, as well as comments, logging, and debugging output. We do not include the pure Eventlets scenario since it implements only part of the Esper functionality. The results are shown in Table 7.2; the Esper E and Esper-Beans implementations have significantly more lines of code than the Eventlet-Esper approach. Further, only two classes are needed for the implementation of Eventlet-Esper. This reduces the complexity of software maintenance tasks compared to Esper E and Esper-Beans.

In contrast to Eventlet-Esper and Esper-Beans the distribution across multiple machines has to be implemented manually in the Esper E scenario. The Esper-Bean scenario uses the Java Enterprise ecosystem, which provides for lifecycle management and distribution. However, since Enterprise Java Beans are not the natural fit to encapsulate event stream processing logic, code has to be adapted in each of the 20 MDBs. This code is referred to as *specific* while *generic* code remains unchanged and is reused. The savings with Esper distributed by means of Eventlets are mainly due to the automated dynamic subscription handling and integrated event handling provided by the Eventlet middleware. Eventlets scale to an arbitrary number of distinct source identifiers without the need for source code modification.

Scenario: Eventlet-Esper

Component	Lines of Code
Eventlet Prototype App. Logic	20
Eventlet Prototype Meta Data	2
Eventlet Registration	1
CEP Queries, Result Handling	18
Total: 2 Classes	**41**

Scenario: Esper E

Component	Lines of Code
Main Class App. Logic	39
Event Listener	10
Distributed Instantiation (JMS-based)	61
CEP Queries, Result Handling	18
Total: 3 Classes	**128**

Scenario: Esper-Beans

Component	Lines of Code
App. logic shared amongst all beans (generic)	19
Bean-specific App. logic	120 (6 lines per bean; 20 beans)
CEP Queries, Result Handling	18
Total: 21 Classes	**157**

Table 7.2: Lines of code comparison of the different implementations used for the evaluation.

7.2 Privacy Preserving Pub/Sub Evaluation

In Section 6.3 we introduced a privacy concept for Eventlets; it relies on a pub/sub infrastructure that performs matching of blinded and encrypted events and subscriptions. The blinding of subscriptions is performed by Eventlet middleware components (Eventlet monitors or Eventlet instances). The matching of events against subscriptions is task of the pub/sub middleware. The matching on encrypted data comes at the cost of an increased computational effort. To quantify this overhead we performed an evaluation: we extended Apache ActiveMQ with our proposed mechanisms for privacy-preserving pub/sub. We extended the subscription matching of ActiveMQ to support matching on blinded message selector attributes in subscriptions. Consumers can issue new subscriptions and register them with the same message listener as the old subscription. The consumers can blind these new subscriptions without interaction with producers or a TTP. We evaluated our implementation and compared it with pub/sub communication without blinding. We did not evaluate the payload encryption as it is evaluated elsewhere [124, 159].

7.2.1 Test Environment and Setup

The test environment is shown in Figure 7.4. We used a distributed setup with two event-generating clients connected to a host running our extended ActiveMQ broker. We started multiple publisher and subscriber threads per machine. When resources of Generator Client I were insufficient, the

second client was used as support. The CPU utilization is plotted accumulatively: at an utilization of approximately 0.35 Generator Client I is fully utilized and Generator Client II is used in addition.

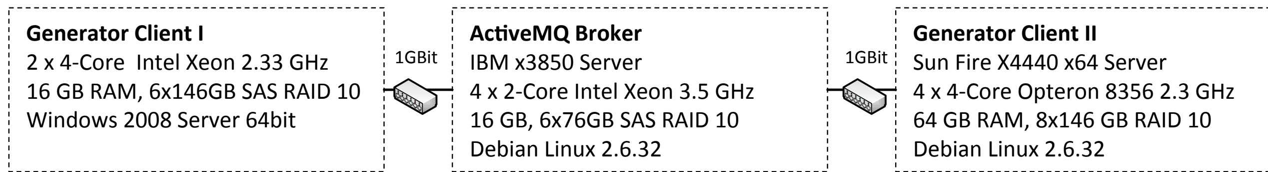


Figure 7.4: Test environment.

In our test case the message selectors used by subscribers required an equality check on an event header attribute (message property). For unencrypted matching the following selector scheme was used: `bigIntProp1 = <Random Big Integer>`. Since our blinding scheme does not allow direct equality checks the following semantically equivalent selector scheme was used for blinded messages: `bigIntProp1 > Blinded(<Random Big Integer>-1) AND bigIntProp1 < Blinded(<Random Big Integer>+1)`. We chose an equality check since this is a common case in the context of Eventlets: an Eventlet instance subscribes for events of a certain entity instance which is, for example, identified by an ID. It is also the worst case for blinded subscriptions. Publishers have to perform two blinding operations per message, subscribers have to perform two blinding operations per subscription, and brokers have to perform two matching calculations per event. A simple *greater than* or *less than* comparison requires half the operations and reduces the overhead compared to unencrypted matching.

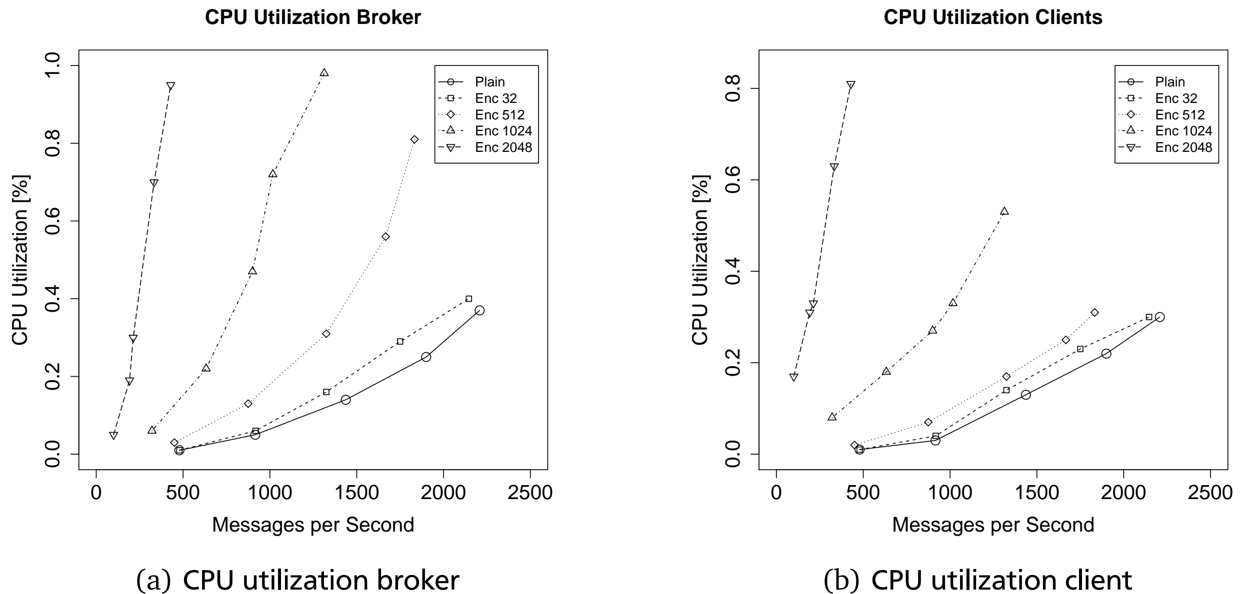
We used random numbers as event payload and to construct subscription filters. The supported domain size for encrypted properties was 100 bits. Inside the broker a matching check was performed for each message. We measured throughput, CPU utilization, and latency for unencrypted messages (plain) and for blinded events with encryption key lengths of 32, 512, 1024, and 2048 bit. We evaluated three scenarios:

- CONSTANT: Constant number of publishers and subscribers per test. Between tests the number of publishers and subscribers was increased resulting in an increased message rate.
- DYNAMIC: Subscribers leave and join the system at a certain rate to simulate context changes and user churn. In the *low dynamics* scenario, 2 subscribers leave and join per second. In the *high dynamics* scenario 20 subscribers leave and join per second. The joins and leaves add additional load to a *static* configuration with a constant rate of subscribers and consumers.
- COMPLEXITY: Subscribers use different message selector lengths. They subscribe with a disjunction of equality checks for up to 6 attribute values. Such a subscription pattern is common, for example, to receive messages of certain types only; messages contain a type attribute which is used for filtering.

7.2.2 Evaluation Results

Figures 7.5(a), 7.5(b), and 7.6 show the results for the CONSTANT scenario. The CPU utilization at broker and client increases with the message rate as well as with the blinding key length. An increase in latency for blinding scenarios was below measurement variance at low loads. With increasing CPU

utilization on broker and clients, the latency increases and shows the typical rapid increase when the systems reach their limit. With a key length of 1024 bit, which can be assumed to be secure, about 1200 messages per second can be handled by the broker. To produce this load, two generator machines are necessary. However, the utilization of the client is only secondary since producers and consumers (Eventlet monitors and Eventlet instances) are typically spread across many machines.



(a) CPU utilization broker

(b) CPU utilization client

Figure 7.5: Results CONSTANT Scenario: CPU utilization for different message rates and blinding strengths

Figures 7.7(a) and 7.7(b) show the CPU demand for the DYNAMIC scenario. Issuing a new subscription requires blinding subscription values on client side. This overhead of blinding new subscriptions is quantified in this evaluation. The CPU demand is shown as CPU time per message in seconds. During the experiments we measured message rate and CPU utilization for the Static, Low -, and High Dynamics configuration. To allow a direct comparison independent of the overall message rate, we show the CPU time per message. The results show that the additional overhead of joining subscribers is not the factor that dominates CPU demand at the broker. Further, the increase occurs independent of blinding, i.e., also in the plain text case. This shows that the overhead of joining is inherent to the broker. On the client side CPU demand increases by about one third in the High Dynamics configuration compared to the Static configuration. No observable difference in the ratio of this increase can be found between the blinded and plain configurations. This shows that the CPU demand of subscription operations is not dominated by blinding operations.

The results for the COMPLEXITY scenario are shown in Figure 7.8; we also use the CPU time per message as metric. For unencrypted subscriptions an increase in CPU time is not observable. For blinded subscriptions the CPU time per message of the broker increases slightly with increasing complexity. The CPU time on the client side increases faster since for each message all attributes have to be blinded, but the broker does not necessarily evaluate the whole message selector. Since the message selector is a disjunction of equality checks a matching subexpression makes the evaluation of other subexpressions obsolete. The steeper increase in client CPU utilization from filter length

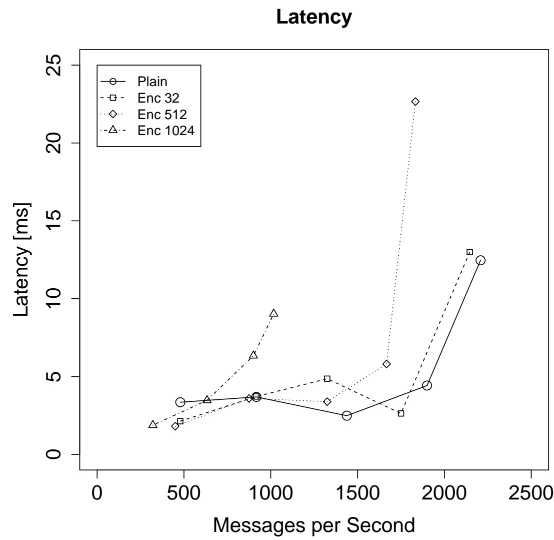
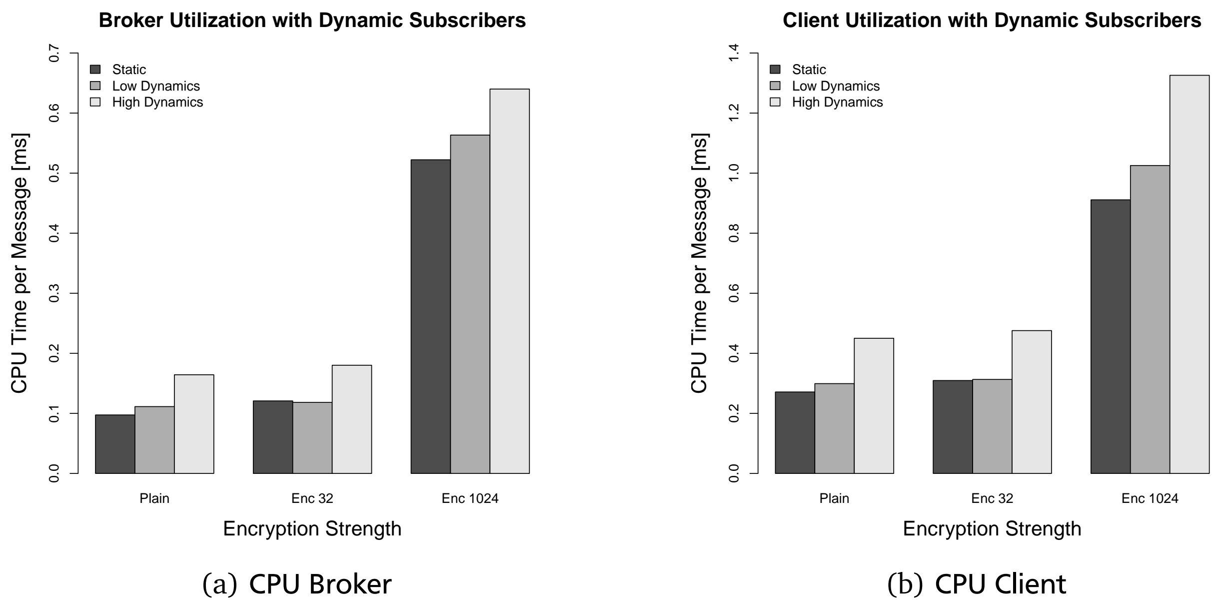


Figure 7.6: Results CONSTANT scenario: Latency for different message rates and blinding strengths.



(a) CPU Broker

(b) CPU Client

Figure 7.7: Results DYNAMIC scenario: Overhead due to frequent subscription updates.

4 to 5 was caused by the activation of the second generator client. The second generator client was necessary to keep the publication rate up. The CPU utilization on this second machine includes basic operating system and Java virtual machine operations, which occur as a one-time effect in the results.

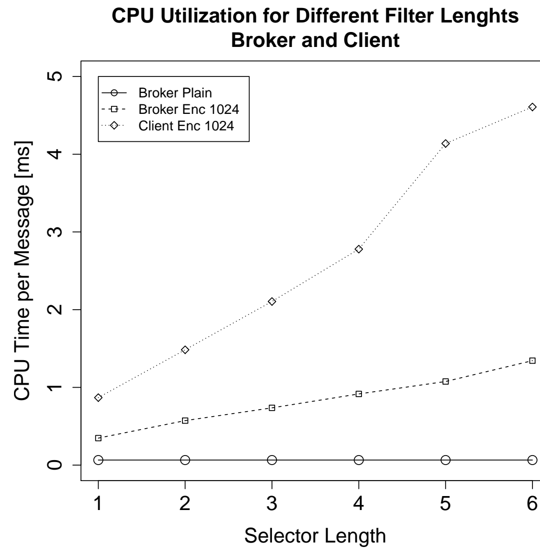


Figure 7.8: Results COMPLEXITY scenario: CPU time required for message dispatch with different message selector lengths.

Overall the evaluation shows that the overhead in terms of CPU utilization of privacy preserving pub/sub is well observable. We also monitored network and memory utilization during the test runs; both were no limiting factors for performance. Thus, the throughput was limited by CPU processing power and privacy preserving pub/sub will benefit from modern multi-core systems with high processing power. It is further possible to extend our scheme and implementation to multiple brokers to build a scalable and secure pub/sub infrastructure. We think the throughput requirements of many applications can then be fulfilled and communication can be secured efficiently by our scheme.

7.3 Summary

The evaluation of our Eventlet middleware shows the advantages with respect to the development of scalable event stream processing applications. Although Eventlets add a layer of abstraction, the performance decrease in direct comparison with a manual implementation is acceptable. Eventlets have the advantages of inherent scalability and ease of software development. We show the necessity of a distributed setup to achieve a high throughput; the required distribution mechanisms are provided by the Eventlet middleware. We also show that the development is simplified with Eventlets; the functionality provided by the Eventlet middleware and the underlying SPU container model significantly reduce the amount of code developers must write.

Further, an important aspect when processing streams of events are privacy concerns, e.g., in health care applications. We thus present the integration of a privacy-preserving pub/sub scheme with Eventlets. To quantify the overhead of privacy-preserving pub/sub we performed an evaluation.

Although the computational overhead is quite significant, it is acceptable in scenarios with sensitive data. Since the cryptographic operations are CPU intensive, modern multi-core CPUs can be used to overcome throughput limitations due to limited computational resources.

8 Related Work

Event processing has become a popular topic and research in many areas addresses challenges related to event processing in different contexts [32, 89]. The focus in our work is on development and integration aspects of event stream processing components. With Event Stream Processing Units (SPUs) we introduce a container model to encapsulate event stream processing in manageable units. SPUs rely on a pub/sub infrastructure for event dissemination. At the conceptual layer, we introduce the SPU container model and present an integration of event-stream processing with business processes by means of SPUs. We also present a requirements engineering approach for event-based systems. At the technical layer, we present and evaluate the Eventlet middleware including privacy and data integration mechanisms. In this chapter we present work related to:

- the conceptual model of SPUs,
- requirements engineering for event-based systems,
- the integration of event stream processing with business processes, and
- the technical realization of our Eventlet middleware.

8.1 Work Related to the SPU Container Model

SPUs share properties with traditional component models [176] as well as with service models [176]. Both, components and services pursue the goal of simplifying application development by introducing modularization and abstraction. Both are important aspect in the evolution process of software [23]. According to Weinrich and Sametinger [176], standardized interfaces, as well as standardized interoperability and execution semantics are the core of component models. This enables the reuse and combination of components into larger applications. SPUs also have a standardized interface and clearly defined execution semantics. SPUs share the loose coupling mechanisms with services [176]: SPU instances are created dynamically and support an implicit invocation scheme. Since SPUs show service as well as software component properties we use the more generic term *container* to categorize SPUs and point out the modularization aspect.

At the conceptual level, SPUs complement Service-oriented Architecture (SOA) services as building blocks for push-based architectures. SOA services encapsulate generic actions with respect to entities and allow for dynamic integration of system components. However, the approach in SOA is pull-based; an application that requires functionality invokes the respective service and waits for the reply. In next-generation SOA the idea of events is integrated to realize reactive services [70, 75, 105]. In Event-driven Service-oriented Architectures (ED-SOAs) [102] and Event-driven Architectures (EDAs) [117, 169], events trigger the invocation of business relevant functionality. In an ED-SOA a service can be invoked by events. In an EDA complex event processing techniques process real-time data and produce events that are consumed by an ED-SOA. An ED-SOA still builds upon services originally designed for request/reply style interaction. With SPUs, we combine ED-SOA aspects and EDA aspects in single containers. We believe that thinking in terms of

generic event-driven tasks is the more natural approach to event-based applications. While a task is something that should happen actively, a service exhibits passive behavior and responds upon request. As result, SPUs follow an implicit invocation scheme while services are invoked explicitly.

In [125], Nastic et al. present a programming model to simplify the development of Internet of Things (IoT) applications in cloud environments. They introduce an abstraction to specify generic tasks that apply to sets of devices. The abstraction concept is similar to the SPU container model but focuses on controlling devices rather than on processing streams of events.

SPUs are also related to mobile software agents [103]. In agent-based systems software components (agents) fulfill tasks autonomously. For example, Bromuri et al. present an approach with distributed agents reacting on events [30]. In their system agents are autonomous proactive components using events to coordinate the overall workflow across all agents. When comparing agent-based systems with SPUs, a single software agent instance is similar to an SPU instance. The SPU concept however is different from agent-based systems; SPU instantiation is event-driven and dynamic depending on the actual events. SPUs are not designed as autonomous units. They rely on a middleware for instantiation and management.

SPUs, along with the concepts and systems discussed before, are language-agnostic. Language-specific approaches integrate event-processing capabilities into programming languages and enable reactive programming. Events are either integrated in existing languages or dedicated reactive languages are used [154]. In EventJava [66], for example, distributed event correlation is seamlessly integrated with methods. In EScala [81], events can be used in an aspect-oriented way in the source code. These extensions make events first class citizens in programming languages and provide event-based functionality without integrating, e.g., Complex Event Processing (CEP) engines. Dedicated reactive programming languages, e.g., AmbientTalk/R [107], make event processing implicitly available. Pub/sub, for example, is used for event dissemination but happens transparently to the programmer who accesses events like other variables. SPUs are also related to active objects [34] like the actor model [4]. Actors are programs that react to incoming events, perform operations depending on the received events, produce outgoing events, and may create further actors. The control flow is solely specified by reactions on events. Examples of actor languages are the Act3 actor language [5] and Scala actors [85].

Compared to SPUs the conceptual goal of language-specific and active-object-based approaches differs. In these approaches events are integral parts of the program code and determine control flow. The focus of SPUs, however, is a data-centric encapsulation of application logic: data in form of event streams is specified as input to containers for application logic where the processing of this data takes place. For this purpose SPUs are designed to be independent from a certain programming language and help to express event-based reactive functionality abstractly. Desired SPU behavior can be expressed using standard programming languages without modifications.

8.2 Related Work to Requirements Engineering in Event-based Systems

Modern information systems underlie frequent changes and often follow a modular architecture. The reuse of single components, however, is hard [79]. An important factor that influences the reuse is the granularity of components [20]. Granularity demands in Event-based Systems (EBSs) determine

the types of events that need to be detected to fulfill certain application demands. With event processing techniques, like event composition, high-level events can be derived from fine granular base events.

The deployment of event detecting components is often a long-term and cost-intensive process, e.g., deployment of wireless sensor nodes at many vehicles. This characteristic of event-based systems requires taking reusability aspects into account during application development: An appropriate requirements engineering process is needed to identify the level of granularity at which events should be detected and published.

The loose coupling in event-based systems makes requirements engineering challenging; loosely coupled components do not necessarily know each other, i.e., a component does not know potential consumers of its output. Goal-based requirements engineering approaches, e.g., Tropos [41] and KAOS [56], are well suited for such dynamic environments. In goal-based requirements engineering the goals an application has to fulfill are specified, e.g., support of a certain use case. The system architecture is then derived to fulfill the specified goals.

The process of deriving a system design from goals can lead to implementation and design alternatives. In [6], Ali et al. present a goal-based requirements engineering approach where different non-functional requirements can be taken into account to evaluate different implementation possibilities.

In our approach presented in Chapter 4 we present a goal-based requirements engineering approach that also takes different implementation alternatives into account and weights them. While the approaches in [6, 41, 56] are generic, our approach is described from a practical perspective with focus on the event-based domain. We start with the identification of required events, i.e., the goal is that the demanded events are available. We then weigh different alternatives of the event detection layer with focus on reusability potential; with a change in the granularity of event detection the event reuse potential increases and more (potential) goals can be fulfilled.

In [94], Jureta et al. build upon goal-based requirements engineering and add, amongst others, Quality of Service (QoS). QoS-aware requirements engineering is relevant for event-based systems as well, as discussed in Section 3.3. It adds another dimension to our approach and should be considered in future work.

8.3 Related Work to Event Processing in Business Processes

The integration of event stream processing with business processes covers three layers: the process modeling layer, the process execution layer, and the IT infrastructure layer. At each layer events and event processing is addressed in research. In [100], Krumeich et al. categorize related research and identify use cases for event processing in the context of business processes. Amongst others, event processing is used to enable flexible process adaption, to monitor cross-organizational processes, to integrate IoT data sources, and to check compliance.

Events are part of various business process modeling notations like Business Process Model and Notation (BPMN) and Event-driven Process Chains (EPCs) [96, 130, 170]; they trigger functions (in EPCs) or tasks (in BPMN) and influence the process control flow. Event processing is often applied

to monitor process execution. Events can be detected by a process execution engine and depict, for example, activation/completion of activities or sending/receiving of messages; events can also be defined implicitly based upon object state changes, e.g., changes in data records [87, 88]. Based upon such events, control flow deviations in single process instances can be detected [175]; it is also possible to check conformance in cross-organizational processes based upon message exchanges [17]. In contrast, our approach focuses on event processing as part of single process instances. The incorporation of (complex) events leads to more reactive and dynamic processes. This is a core concept in EDAs [44, 117] or ED-SOA [105]. However, event streams do not have explicit representations in BPMN or EPCs. Currently, event streams have to be modeled explicitly as multiple events, e.g., using loops that process events. Such explicit modeling of complex events and event processing is for example presented in [18, 26, 61, 178]. In [39, 167], BPMN is extended to allow modeling of tasks implemented with wireless sensor networks; these extensions share some aspects with event stream processing, but do not support the full semantics of implicit/explicit instantiation/completion. In [26], Biörnstad et al. discuss different alternatives for the integration of event streams with processes: the consumption of events can be modeled explicitly (process-driven) as loop in the process where events are pulled from a source regularly. The integration of events can also be in a push style and source-driven; this is the approach we follow with SPUs.

Process models are often created by business experts without detailed knowledge about technical details of event processing. Further, to make models intuitively understandable, modelers should use as few elements as possible with self-explaining activity labels [115]. Thus, activities should represent business functions. Services are a successful abstraction mechanism to support this. Services represent business functions and exhibit a data input/output interface [137]. Process models do not (and should not) contain the application logic of a service; this is left to service developers who can use more appropriate modeling notations to describe the technical details. Thus, our approach confers basic service concepts [59] to event stream processing and introduces SPUs as an appropriate abstraction. We concentrate on control flow oriented business process models represented with BPMN and EPCs. However, the SPU concept is also applicable to alternative approaches for managing business operations and processes, e.g., the Guard-Stage-Milestone (GSM) approach, which allows a more declarative modeling of business activities [92]. In GSM, SPUs can be represented by stages. Instantiation is modeled as guards: a stage is activated when an event stream becomes available (implicit instantiation guard) or when an event triggers the start of event stream processing (explicit instantiation guard). Completion is modeled as milestones: an implicit completion milestone holds the SPU completion condition; and explicit completion milestone waits for an event published by other stages.

At the process execution layer, Barros et al. [18] evaluate service interaction pattern support in BPMN and Business Process Execution Language (BPEL); many patterns cannot be realized with BPMN or BPEL due to inappropriate event support, e.g., consumption policies. We thus encapsulate event stream processing functionality in Event Stream Processing Tasks (ESPTs) and provide an interface in our Eventlet middleware to control the execution of ESPTs; the event stream processing as such, e.g., event consumption, is then transparent to the process execution language.

Juric [95] presents extensions to BPEL that allow service invocations by events. In [84], Spiess et al. encapsulate event sources as services. Both approaches do not address event streams as

input/output to/from components; rather than a stream of events, single events are understood as business relevant.

At the technical layer of process execution, event streams are well known. CEP is supported by a variety of tools, e.g., the Esper CEP engine, Software AG Apama, or IBM InfoSphere Streams. CEP is also part of business process execution environments like JBoss jBPM/Drools. In [101], business process modeling techniques are used to express CEP queries. Event stream processing is integrated bottom-up; CEP queries and rules are specified at the technical layer. In contrast, our approach is top-down and business entity centric event streams are visible as input/output of ESPTs at the modeling layer. Event streams can be as business relevant as, e.g., input/output data of services. Thus, like service task input/output is explicit in models, event streams are explicit at the modeling layer in our approach.

The event stream processing application logic inside Eventlets can be simple rules, CEP queries, or complex event processing networks as described in [62]. Our middleware instantiates Eventlets for each entity instance, e.g., one CEP query is issued per shipment. This encapsulation of event stream processing logic is related to *design by units* described in [168]. It improves scalability and fosters elasticity.

8.4 Related Work to Eventlets

Work related to Eventlets (as implementation of the SPU concept) addresses implementation aspects of event processing. Architectures and models for push-based reactive software systems in general have been addressed in previous work; in [53] the authors present a survey where distributed event-based architectures are described from different points of view. In [27], Blanco et al. introduce a meta-model for distributed event-based systems based on a peer-to-peer system. Their system shares the idea of reactive components with Eventlets. However, they do not introduce a high level of abstraction with a generic view on tasks.

According to the taxonomy of distributed event-based systems presented in [113], our Eventlet middleware implements an implicit event model. It hides subscription details from developers who only specify filters on the event content in Eventlet prototypes. The Eventlet middleware as such builds on top of a mediator/broker network with "functionally equivalent" brokers. Our Eventlet middleware is also a "distributed service with separated multiple middleware", i.e., Eventlet instances can run on different machines and consume events from a distributed broker network.

A comparison of Eventlets with the software component models presented in [104] shows that Eventlets share technical properties with Enterprise Java Beans (EJBs), especially Message-driven Beans (MDBs). However, EJBs are not the best fit for scalable event stream processing in terms of performance and ease of development as we show in our evaluation (cf. Section 7.1). In EJBs, messages were introduced as asynchronous inter-component communication mechanism and MDBs statically define their subscriptions at compile time. Eventlets, in contrast, issue subscriptions dynamically at runtime depending on the event stream.

Event-condition-action (ECA) rules are another mechanism to implement reactive applications [43, 58]. With dynamic ECA rule replication and generic rules it is possible to generalize the action part of ECA rules and implement SPU-like semantics: for each SPU instance a corresponding ECA

rule would be required with a condition that filters entity instance specific events. The action part is then identical for all rules. However, this requires management components for rule replication and interpretation of generic expressions. In addition to pure reactive behavior expressed with ECA rules, Eventlets provide mechanisms for lifecycle management. Upon instantiation and removal of Eventlet instances additional application logic can be integrated and the validity of Eventlet instances is specified explicitly and checked at runtime.

Another area of related research is complex event processing. CEP engines, like Esper or IBM InfoSphere Streams, are typically integrated into an event-based infrastructure by connecting them to the event bus (see Section 2.1.3). SPUs can implement CEP functionality on their own as application logic. SPUs can also integrate existing CEP solutions when required: upon instantiation of an SPU a CEP query can be issued, as shown in our evaluation (see Section 7.1). With the integration of SPUs and CEP, generic complex event queries can be realized in a distributed way.

SPUs build on top of a pub/sub system with content-based pub/sub capabilities. Pub/sub systems as such can also provide event processing functionality, e.g., for event composition [16]. Subscription filter expressions are then more expressive and contain CEP query elements. This can be used to implement SPUs that are instantiated based upon complex conditions as described in Section 3.5. In general, SPUs decouple event processing from the pub/sub layer; this allows a loose coupling and is advantageous in terms of scalability [142]. In addition to this loose coupling, SPUs can be instantiated implicitly; this form of implicit invocation further simplifies the development of event processing applications [80].

In Sections 6.3 and 6.4 we discuss privacy and data integration aspects in the Eventlet middleware. Privacy demands in pub/sub systems have been identified by Chenxi et al. in [174] and addressed by researchers. In [144], for example, Raiciu et al. present a privacy-preserving brokering scheme; in contrast to the scheme we applied for Eventlets, false positives may occur and events are delivered to wrong subscribers. The privacy-preserving brokering scheme presented by Srivatsa et al. in [161] requires a TTP contact for each subscription; we avoid such a contact upon resubscription for better scalability.

In event-based systems participants are loosely coupled and interact anonymously; this requires data integration approaches tailored to such environments [50]. On the one side of the spectrum, self-describing approaches can be applied where semantical information is attached to each event [28]. On the other side an ontology can be specified a priori and transformation rules can be derived based upon this ontology [164,173]. The approach we suggest for Eventlets lies in between: with the specification of transformation rules at subscription an ontology is specified implicitly and semantical description is kept apart from the event data itself.

8.5 Summary

The event-based paradigm is widely addressed by researchers. In many application domains event-based technologies are applied to implement reactive behavior. One major field in event-based research is related to event stream processing; this is the area of our work. Characteristic to event stream processing is that sets of homogeneous events (streams) are considered. For the dissemination of such event streams pub/sub technologies are appropriate. The use of event streams in

business applications complements the traditional request/reply interaction scheme in SOAs and allows for reactive applications that integrate real-world information. Adapted requirements engineering approaches are required to take this new type of data into account since event production and consumption is decoupled. Existing integration concepts of the event-based approach with the business layer (business process modeling and execution) focus on single events rather than on event streams; our approach provides an integration concept for event streams. Overall, a lot of related research addresses specific technological challenges. We focus on a model to simplify the integration of event stream processing with applications; this model depends upon many aspects of the research at the technological level. The better the underlying event processing techniques are, the more opportunities exist to leverage the information provided in form of events.

9 Conclusion

In today's world the pervasive use of mobile devices and sensors provides real-time information of small granularity in form of event streams. In addition, events occur at a large scale in many software systems, e.g., postings in social networks or stock price notifications. These event streams reflect the state of the real world and foster reactive behavior in software systems. Patient monitoring, smart home environments, as well as traffic management systems are typical examples [52, 86, 89]. In contrast to the traditional handling of data, event streams provide information in a push-based manner with a complete decoupling between event producers and consumers. Thus, mechanisms like pub/sub and implicit invocation are essential to process these event streams efficiently. However, integrating such event stream processing in existing application landscapes and business processes is challenging: scalability, modularity, reusability, and manageability are important properties. To foster these properties, application logic needs to be encapsulated and requirements engineering needs to be aware of event streams. In this work we present an appropriate encapsulation and requirements engineering concept for general-purpose event stream processing tasks. With our concept of Event Stream Processing Units (SPUs) we provide appropriate building blocks in form of a container model: SPUs are suitable for developing stand alone scalable event stream processing applications as well as for the integration of event stream processing in the enterprise application context, i.e., in business process modeling and business process execution. Our event-stream-aware requirements engineering process enables reusability assessment early in the development process.

SPUs, as introduced in Section 3.2, provide an intuitive abstraction to encapsulate event stream processing in a generic way. SPUs have a managed lifecycle and can be instantiated implicitly and explicitly. Since events arrive continuously, a request/reply interaction mode is not applicable. Event processing is a continuous operation and the completion of SPUs needs to be triggered; this can happen implicitly or explicitly. For instantiation and completion, SPUs require meta data in form of constant and instantiation expression; based upon those expressions, subscriptions for single SPU instances are derived, e.g., a subscription for a specific shipment. With explicit instantiation the value set that uniquely identifies the event stream associated with a specific entity instance is provided externally. With implicit instantiation the SPU execution environment ensures that for each distinct instantiation value set, e.g., for each shipment ID, an SPU instance is created. For implicit completion a validity expression needs to be specified; this expression is evaluated internally by the SPU. When the expression evaluates to true, the SPU completion can be triggered. SPUs implement methods required for lifecycle management: `onInstantiation`, `onRemove`, and `onExpiration`. Application logic associated with the creation and removal of SPU instances as well as with implicit completion is located in these methods. The `onEvent` method is called each time an event arrives and is the toehold for SPU-internal event dispatch.

With the distinction between meta data and runtime code, SPUs provide a clear separation between subscription logic and application logic. In addition, SPUs do not introduce a new programming language; they rather provide an abstraction layer on top of a pub/sub system. Automatisms for

connection management, i.e., dynamic creation of subscriptions, and distribution allow an object-oriented development process for scalable event stream processing applications.

Since SPUs rely on a pub/sub system, the Quality of Service (QoS) of the underlying event dissemination infrastructure is crucial for the QoS provided by SPUs. We introduce various QoS aspects related to the interface between the pub/sub system and SPUs in Section 3.3. SPUs, as event consumers, depend on the performance of the pub/sub systems as well as on its reliability properties. Further, scalability is an important aspect to guarantee a certain QoS under different loads. We show how SPUs can interact to realize a scalable event stream processing application (see Section 3.4). Since event streams are partitioned into entity instance specific sub streams, SPUs have inherent distribution capabilities. SPU instances run independently of each other and can be deployed in a distributed setting. In terms of enterprise application integration SPUs are a push-based equivalent to services: services are the foundation for SOAs, while SPUs are the foundation for EDAs. Just like services are integrated in a service-oriented architecture to model workflows in a function-centric way [105], SPUs can be integrated in an event-based infrastructure to implement complex reactive functionality.

With Eventlets, we present an implementation of the SPU container model on the basis of Java (see Chapter 6). Like web services are a technology to implement services, Eventlets are a technology to implement SPUs. Eventlets are represented by Java objects that inherit functionality for lifecycle management and implicit/explicit instantiation/completion. We provide an adapter for the Java Message Service (JMS) as pub/sub middleware and support XML as well as att/val-based events. Our Eventlet middleware is capable of distributing Eventlet instances across multiple machines. We also show how a privacy concept can be integrated with our middleware to ensure confidentiality of event producers and consumers. To handle heterogeneity we show the application of our event transformation approach located inside the middleware. In our evaluation we quantify the benefits of Eventlets in terms of scalability and simplified software engineering (see Section 7.1). We show that the inherent distribution capabilities of SPUs, respectively Eventlets, are required for scalable event stream processing applications. The modular design of the Eventlet infrastructure enables distribution across multiple nodes. Our evaluation shows that this distribution is necessary for scalable event-based applications. We further determine the performance overhead of encapsulating a traditional distributed CEP application by means of Eventlets. Compared to a manual implementation of a distributed event stream processing application the overhead introduced by the Eventlet middleware is negligible given those advantages in term of system development and management. We also quantified the overhead for applying privacy-preserving encryption techniques to Eventlets; depending on the use case the performance decrease is acceptable given the gained privacy (see Section 7.2).

The encapsulation of application logic is also a prerequisite for the integration of event stream processing with business processes management and enterprise architectures. The goal of business processes management is the structured description of business activities. This is achieved by identifying business operations and their interdependencies; the resulting business process models are a stepwise representation of business cases. Single business operations or business functions are self-contained units with defined input, output, and semantics. SPUs can represent such business functions; their properties make them suitable building blocks for the integration of event stream

processing with business processes (see Section 5.1). At the abstract layer, SPUs are integrated into business process models where they act as abstract representation of event stream processing. We present extensions of Event-driven Process Chains (EPCs) and Business Process Model and Notation (BPMN) to include SPUs in process models. After processes have been modeled, the next step is an automated execution. Processes modeled with EPCs need to be mapped to technical process representations in BPMN as shown in Section 5.2.1. Afterwards, SPUs are brought to execution (see Section 5.2.2). Implicit and explicit instantiation/completion semantics are represented in an executable workflow representation. During execution, our Eventlet middleware interfaces with the process execution environment and acts as runtime environment for SPUs, which are implemented as Eventlets (see Section 5.2.3). We implemented the modeling and transformation of SPU-including process models with Software AG products to show the applicability in industrial applications (see Section 5.3).

From the enterprise architecture perspective, SPUs are a push-based equivalent to services. Traditional services are not the natural environment for event stream processing application logic; current enterprise architectures are not designed to support event stream specific execution semantics like implicit instantiation and completion as well as asynchronous and concurrent reactions. We thus introduce SPUs as a software concept to map the real world concept of generic event-driven tasks seamlessly. With SPUs it becomes possible to think event-based from the very beginning of application development rather than adding event-based capabilities to a pull-based system architecture. SPUs are not meant to replace SOA services, but to enable reactive functionality in scenarios for which SOA is not a natural fit [112].

One of the main characteristics of event-based systems is that event production is decoupled from event consumption (see Section 2.1.4). Further, the reuse of events, i.e., a high fan out during pub/sub event dissemination, is desirable. This needs to be taken into consideration throughout the development process of event-based applications; this development process starts with the requirements engineering. Since it cannot be assumed that demanded events are already present, the question of availability of events is an essential part in the requirements engineering process. In our SPU container model, for example, we demand entity-instance-centric events that allow an object-like view for the development. Availability of demanded events along with a high reuse potential are addressed in our requirements engineering approach for event-based systems (see Chapter 4). First, required events are identified (see Section 4.4). Our methodology then allows the comparison of alternative event generation approaches based upon the reuse potential. We distinguish between derivable events and base events (see Section 4.1). Derivable events can be generated with CEP mechanisms, e.g., event composition. They also can be detected directly, e.g., with specialized sensors. Our approach discusses the different alternatives; with costs assigned to the different event production alternatives, different solutions can be compared. Generally, the more fine grain events are, the higher are the chances for reuse in other contexts. However, detecting many fine grain events to compose a required events results in increased effort and costs.

Although our approach helps to decide upon which events to detect directly and which events to compose, the major problem is the estimation of the potential for future reuse of events. Implementing an event detection and composition infrastructure based upon many base events with only one consumer for the composed events is not efficient. An essential part in the development process

of event generating components is thus the identification of potential use cases that require events. Only then event generation can be optimized for reusability and costs can be saved. For example, long and cost intensive deployment cycles of event detecting hardware can be avoided by deploying hardware with more capabilities in first place even though all capabilities are not yet needed.

Summarizing our work, we simplify the development of scalable and distributed event stream processing applications by means of the SPU container model. We present Eventlets, an implementation of our SPU container model, along with a distributed middleware. We address the specifics of Event-based Systems (EBSs) development in our requirements engineering approach: we foster the reuse of events in arbitrary event-based components, e.g., in different SPUs. We further integrate event stream processing with business process modeling and execution. We introduce extensions to EPCs and BPMN and describe the model-to-execute workflow that brings SPU-containing business process models to execution.

10 Future Research

Future research related to Event Stream Processing Units (SPUs) has two directions: research related to the conceptual model of SPUs and research related to technical aspects of SPU execution. Future research at the conceptual layer addresses interaction and integration aspects of SPUs and the requirements engineering methodology. Future research directions at the technical layer address the SPU implementation and runtime infrastructure, i.e., Eventlets and the Eventlet middleware, as well as the execution of SPUs in the context of business processes.

10.1 Conceptual Layer

SPU Interaction and Integration Concepts

Like services and object-oriented programming, SPUs are an abstraction that enhances the development of event stream processing software systems. The conceptual foundation – our SPU container model – is suitable to modularize event stream processing application logic. However, it is not advisable to apply the SPU concept to each event stream processing scenario. In use cases where an entity-instance-centric partitioning of event streams is not feasible, different concepts have to be applied. Monitoring a shipment, for example, is well suited for implementation with SPUs since event processing is entity instance centric, i.e., the application logic can be defined on a per-shipment basis. Calculating the average delivery time over all shipments, in contrast, can be implemented with a single Complex Event Processing (CEP) query; an implementation with SPUs would result in a single SPU instance since the task is not entity instance centric. In terms of future research such design decisions are an area of interest. Event stream processing applications can be analyzed to identify tasks that can be expressed by entity instance centric application logic. A categorization and grouping of such tasks can then be used to derive a design guideline with typical patterns where to use SPUs to encapsulate event stream processing application logic.

Our approach for integrating event stream processing with business processes by means of SPUs is also an area for future research. SPUs in general and Event Stream Processing Services (ESPSs) and Event Stream Processing Tasks (ESPTs) in particular depict tools that enable modeling event stream processing in business processes. Developing a comprehensive understanding and description of the resulting execution semantics, however, is challenging. SPUs issue subscriptions; but successfully issued subscriptions do not guarantee that demanded events are published. Further, business process models can describe complex situations, which require, for example, transactional behavior. In such cases SPUs require compensation functionality to support rollbacks although event producers and consumers are logically decoupled.

SPUs enable the combination of push- and pull-based interactions within business processes. Thus, the interplay of push- and pull-based components in the context of large and complex processes is a topic of further investigation. The analysis of SPU interaction and integration patterns can result in generic patterns, e.g., comparable to service interaction patterns [19]. In cross-organizational

processes aspects like data integration and quality of service at the IT infrastructure layer require novel solutions. At the business process modeling layer, meta-model extensions that describe SPUs are necessary to provide a basis for interoperability.

Novel Requirements Engineering Approaches

Our approach of requirements engineering for event-based applications can be used in traditional software development processes where an assessment of requirements is the first development step, e.g., software development that follows the V-Model. Future research can address the applicability of our approach in the context of agile software development. Since requirements engineering is an iterative process in agile methods, e.g., in Scrum where the product backlog can be extended throughout the development process, an a priori assessment of demanded events and possible reuse potential is challenging. It could be evaluated under which circumstances agile methods can or cannot be applied to develop event-based applications. The deployment of sensor hardware, for example, can be a time consuming process and might not fit well with the short development-to-feedback cycles applied in agile methods. In case events sources are already present, however, agile methods might be well suited for the development of event-based applications.

10.2 Technical Layer

Eventlets and Eventlet Middleware

With Eventlets and the Eventlet middleware we present an implementation of SPUs. Our Eventlet middleware is designed to run in a distributed setting; Eventlet instances are self-contained and receive events independently. An issue that originates from the distributed design and that can be addressed in future research is a guaranteed lossless Eventlet instantiation process, i.e., no events are lost during the creation and subscription process of an Eventlet instance. When an Eventlet monitor triggers the creation of an Eventlet instance, time elapses until the newly created Eventlet instance receives events. However, use cases might exist where it needs to be guaranteed that events occurring during this instantiation process are not lost. One way to achieve this is early explicit instantiation of Eventlets; Eventlet instances are created prior to the occurrence of events associated with the Eventlet instance, e.g., a shipment monitoring Eventlet instance is created before the shipment is assembled. However, explicit instantiation is not always feasible since external knowledge about (expected) events might not be available, e.g., which cars enter a city and need to be tracked for toll processing. Thus, implicit instantiation is based upon implicit invocation. To ensure a lossless Eventlet instance creation process with implicit instantiation, events need to be cached. One possibility is the implementation of caching inside the Eventlet monitor. Upon receiving the event that triggers the creation of an Eventlet instance, consecutive events associated with this instance are cached by the Eventlet monitor. As soon as the Eventlet instance is ready to process events, a handover procedure is initiated and cached events are replayed to the Eventlet instance. As soon as the replay of events is in sync with the currently arriving events, the Eventlet monitor stops caching.

Another possibility for guaranteed loss-less instantiation is implementing caching capabilities at the pub/sub layer and to allow subscriptions to the past. With such a pub/sub system the Eventlet

monitors record the timestamp of the event that triggers the creation of an Eventlet instance. The Eventlet instance then issues a subscription for events back to this timestamp. The pub/sub system then replays old events and handles the handover between archived events and current events. This happens transparently to subscribers like Eventlet instances.

Event caching mechanisms are also the basis for migration of Eventlet instances. Currently, Eventlet instances are bound to the network node of their creation. However, Eventlet instances could be moved between nodes; this is especially applicable in cloud infrastructures to enable an efficient scaling (up and down). Eventlet instances are then moved between virtual machines as the demand of computational resources increases/decreases. To reduce network traffic, Eventlet instances can be moved to their corresponding Eventlet monitor, or close (in terms of network topology) to the producers of events. An event caching mechanism is required to bridge the time of state migration. Event consumption needs to be stopped and the current state of the Eventlet instance needs to be transferred to an Eventlet instance located at another node. Mechanisms required for Eventlet instance migration can also act as a foundation for failover strategies; when the crash of an Eventlet instance is detected, events could be cached and handed over to a newly created Eventlet instance. Similar concepts exist in Service-oriented Architectures (SOAs), e.g., service continuation mechanisms [165].

Future research might also address large-scale deployments of the Eventlet middleware. In large-scale deployments, Eventlet monitors may become a bottleneck: an Eventlet monitor receives all events corresponding to its constant expression, i.e., all events for which an Eventlet instance should exist. This 1:n relationship between Eventlet monitors and their associated Eventlet instances might limit the overall performance. One possibility to overcome this issue is breaking the 1:n relationship and using multiple Eventlet monitors to trigger the creation of Eventlet instances. Such a load balancing can be realized by splitting Eventlet monitor subscriptions. In the shipment monitoring use case, for example, the range of shipment IDs can be split and sub ranges can be assigned to different Eventlet monitors. Each of these monitors adds the according filter parameters to its subscription, e.g., shipment ID between 0 and 999. With such filters an Eventlet monitor is then only responsible for the instantiation of Eventlets in the filter range. However, additional knowledge is required to implement this load balancing strategy: the range of shipment IDs and the distribution across this range needs to be known.

Further, scalable pub/sub systems are also required for large-scale deployments of the Eventlet middleware. Additional adapters to integrate distributed pub/sub systems with sophisticated routing mechanisms can be added to our middleware to ensure that the event dissemination does not become a bottleneck.

Business Process Modeling and Execution

In the context of executing SPU-containing process models, the automated model transformation process can be optimized to support a seamless model-to-execute workflow. Currently, manual refinements are necessary during the transition from an abstract to an executable process model representation. To reduce the effort in the manual refinement process, a validity check component can be integrated in the modeling and transformation process to support the creation of well-executable models. Such a validity checker ensures that abstract process models follow certain rules so that the

executable process is well formed. Examples for such rules are checking explicitly instantiated SPUs for corresponding stop signals on all control flow paths or whether signals emitted from SPUs are caught and handled.

Bibliography

[1] Wil M.P. Aalst, Arthur H.M. Hofstede, and Mathias Weske. Business process management: A survey. In *1st International Conference on Business Process Management (BPM)*, The Netherlands, 2003.

[2] Raman Adaikkalavan and Sharma Chakravarthy. SnoopIB: Interval-based event specification and detection for active databases. *Elsevier Journal: Data & Knowledge Engineering*, 59(1):139–165, 2006.

[3] Sebastian Adam, Norman Riegel, Joerg Doerr, Oezguer Uenalan, and Daniel Kerkow. From business processes to software services and vice versa – an improved transition through service-oriented requirements engineering. *Wiley Journal of Software: Evolution and Process*, 24(3):237–258, 2012.

[4] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1985.

[5] Gul Agha and Carl Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In *5th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, India, 1985.

[6] Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. A goal-based framework for contextual requirements modeling and analysis. *Springer Journal: Requirements Engineering*, 15(4):439–458, 2010.

[7] AMQP Working Group. *Advanced Message Queuing Protocol Standard 1.0*. Organization for the Advancement of Structured Information Standards (OASIS), 2010.

[8] Jose Antollini, Mario Antollini, Pablo Guerrero, and Mariano Cilia. Extending Rebeca to support concept-based addressing. In *1st Argentine Symposium on Information Systems (ASIS)*, Argentina, 2004.

[9] Stefan Appel, Sebastian Frischbier, Tobias Freudenreich, and Alejandro Buchmann. Eventlets: Components for the integration of event streams with SOA. In *5th IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, Taiwan, 2012.

[10] Stefan Appel, Sebastian Frischbier, Tobias Freudenreich, and Alejandro Buchmann. Event stream processing units in business processes. In *11th International Conference on Business Process Management (BPM)*, China, 2013.

[11] Stefan Appel, Pascal Kleber, Sebastian Frischbier, Tobias Freudenreich, and Alejandro Buchmann. Modeling and execution of event stream processing in business processes. *Elsevier Journal: Information Systems*, 2014.

[12] Stefan Appel and Kai Sachs. A logistics workload for event notification middleware. In *From Active Data Management to Event-Based Systems and More*, volume 6462. Springer Lecture Notes in Computer Science (LNCS), 2010.

[13] Stefan Appel, Kai Sachs, and Alejandro Buchmann. Quality of service in event-based systems. In *22nd GI-Workshop on Foundations of Databases (GvD)*, Germany, 2010.

[14] Stefan Appel, Kai Sachs, and Alejandro Buchmann. Towards benchmarking of AMQP. In *4th ACM International Conference on Distributed Event-Based Systems (DEBS)*, UK, 2010.

[15] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In *30th International Conference on Very Large Data Bases (VLDB)*, Canada, 2004.

[16] Jean Bacon, John Bates, Richard Hayton, and Ken Moody. Using events to build distributed applications. In *2nd International Workshop on Services in Distributed and Networked Environments*, Canada, 1995.

[17] Aymen Baouab, Olivier Perrin, and Claude Godart. An optimized derivation of event queries to monitor choreography violations. In *10th International Conference Service-Oriented Computing (ICSOC)*. China, 2012.

[18] Alistair Barros, Gero Decker, and Alexander Grosskopf. Complex events in business processes. In *10th International Conference on Business Information Systems (BIS)*, Poland, 2007.

[19] Alistair Barros, Marlon Dumas, and ArthurH.M. Hofstede. Service interaction patterns. In *3rd International Conference on Business Process Management (BPM)*, France, 2005.

[20] Victor R. Basili and Hans Dieter Rombach. Support for comprehensive reuse. *IEEE Software Engineering Journal*, 6(5):303–316, 1991.

[21] Jörg Becker, Michael Rosemann, and Christoph Uthmann. Guidelines of business process modeling. In Wil Aalst, Jörg Desel, and Andreas Oberweis, editors, *Business Process Management*, volume 1806 of *Lecture Notes in Computer Science*, pages 30–49. Springer, 2000.

[22] Stefan Behnel, Ludger Fiege, and Gero Mühl. On quality-of-service and publish/subscribe. In *5th International Workshop on Distributed Event-based Systems (DEBS)*, Portugal, 2006.

[23] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *22nd International Conference on Software Engineering (ICSE)*, Ireland, 2000.

[24] Shimshon Berkovits. How to broadcast a secret. In *Workshop on the Theory and Application of Cryptographic Techniques (EUROCRYPT)*, United Kingdom, 1991.

[25] Philip Bianco, Grace A. Lewis, and Paulo Merson. Service level agreements in service-oriented architecture environments. Technical Report CMU/SEI-2008-TN-021, Software Engineering Institute, Carnegie Mellon University, 2008.

[26] Biörn Biörnstad, Cesare Pautasso, and Gustavo Alonso. Control the flow: How to safely compose streaming services into business processes. In *3rd IEEE International Conference on Services Computing (SCC)*, USA, 2006.

[27] Rolando Blanco, Jun Wang, and Paulo Alencar. A metamodel for distributed event based systems. In *2nd ACM International Conference on Distributed Event-based Systems (DEBS)*, Italy, 2008.

[28] Christof Bornhövd. Semantic metadata for the integration of Web-Based data for electronic commerce. In *1st International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS)*, USA, 1999.

[29] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. Cayuga: a high-performance event processing engine. In *ACM International Conference on Management of Data (SIGMOD)*, China, 2007.

[30] Stefano Bromuri and Kostas Stathis. Distributed agent environments in the ambient event calculus. In *3rd ACM International Conference on Distributed Event-Based Systems (DEBS)*, USA, 2009.

[31] Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer Journal*, 20(4):10–19, 1987.

[32] Manfred Broy, María Victoria Cengarle, and Eva Geisberger. Cyber-physical systems: Imminent challenges. In Radu Calinescu and David Garlan, editors, *Large-Scale Complex IT Systems. Development, Operation and Management*, volume 7539 of *Lecture Notes in Computer Science*. Springer, 2012.

[33] Ralf Bruns and Jürgen Dunkel. Towards pattern-based architectures for event processing systems. *Wiley Journal of Software: Practice and Experience*, 2013.

[34] Alejandro Buchmann. Modeling heterogeneous systems as an active object space. In *4th International Workshop on Persistent Object Systems (POS)*, USA, 1990.

[35] Alejandro Buchmann, Stefan Appel, Tobias Freudenreich, Sebastian Frischbier, and Pablo E. Guerrero. From calls to events: Architecting future BPM systems. In *10th International Conference on Business Process Management (BPM)*, Estonia, 2012.

[36] Alejandro Buchmann, Christof Bornhoevd, Mariano Cilia, Ludger Fiege, Felix Gaertner, Christoph Liebig, Matthias Meixner, and Gero Muehl. Dream: Distributed reliable event-based application management. In Mark Levene and Alex Poulovassilis, editors, *Web Dynamics: Adapting to Change in Content, Size, Topology and Use*. Springer, 2004.

[37] Alejandro Buchmann, Hans-Christian Pfohl, Stefan Appel, Tobias Freudenreich, Sebastian Frischbier, Ilia Petrov, and Christian Zuber. Event-Driven services: Integrating production, logistics and transportation. In *8th International Conference on Service Oriented Computing (ICSOC) Workshops*, USA, 2010.

[38] Business Process Management Initiative (BPMI). *Business Process Model and Notation (BPMN), Version 1.0*. 2004.

[39] Alexandru Caracaş and Thorsten Kramp. On the expressiveness of BPMN for modeling wireless sensor networks applications. In *3rd International Workshop on Business Process Model and Notation (BPMN)*, Switzerland, 2011.

[40] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *19th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, USA, 2000.

[41] Jaelson Castro, Manuel Kolp, and John Mylopoulos. Towards requirements-driven information systems engineering: the Tropos project. *Elsevier Journal: Information Systems*, 27(6):365–389, 2002.

[42] Sharma Chakravarthy and Qingchun Jiang. *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing*. Springer, 2009.

[43] Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *20th International Conference on Very Large Data Bases (VLDB)*, Chile, 1994.

[44] Payal Chakravarty and Munindar P. Singh. Incorporating events into cross-organizational business processes. *IEEE Internet Computing Journal*, 12(2):46–53, 2008.

[45] K. Mani Chandy. Sense and respond systems. In *31th International Computer Measurement Group Conference (CMG)*, USA, 2005.

[46] K. Mani Chandy, Michel Charpentier, and Agostino Capponi. Towards a theory of events. In *1st International Conference on Distributed Event-Based Systems (DEBS)*, Canada, 2007.

[47] K. Mani Chandy and W. Roy Schulte. *Event Processing: Designing IT Systems for Agile Companies*. McGraw-Hill, Inc., 2010.

[48] Mani K. Chandy, Opher Etzion, and Rainer von Ammon. 10201 executive summary and manifesto – event processing. In K. Mani Chandy, Opher Etzion, and Rainer von Ammon, editors, *Event Processing*, number 10201 in Dagstuhl Seminar Proceedings, Germany, 2011. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[49] Guang-Huei Chiou and Wen-Tsuen Chen. Secure broadcasting using the secure lock. *IEEE Transactions on Software Engineering (TSE)*, 15(8):929–934, 1989.

[50] Mariano Cilia, Mario Antollini, Christof Bornhövd, and Alejandro Buchmann. Dealing with heterogeneous data in pub/sub systems: The concept-based approach. In *3rd International Workshop on Distributed Event-Based Systems (DEBS)*, UK, 2004.

[51] Kevin Cline, Josh Cohen, Doug Davis, Donald F Ferguson, Heather Kreger, Raymond McCollum, Bryan Murray, Ian Robinson, Jeffrey Schlimmer, John Shewchuk, et al. Toward converging web service standards for resources, events, and management. *A Joint White Paper from Hewlett Packard Corporation, IBM Corporation, Intel Corporation and Microsoft Corporation*, 2006.

[52] Diane J. Cook and Sajal K. Das. *Smart Environments: Technology, Protocols and Applications*. Wiley, 2005.

[53] Valentin Cristea, Florin Pop, Ciprian Dobre, and Alexandru Costan. *Distributed Architectures for Event-based Systems*, volume 347 of *Studies in Computational Intelligence (SCI)*. Springer, 2011.

[54] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering (TSE)*, 27:827–850, 2001.

[55] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15:1–15:62, 2012.

[56] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Elsevier Journal: Science of Computer Programming*, 20(1–2):3–50, 1993.

[57] Umeshwar Dayal, Barbara Blaustein, Alejandro Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record*, 17(1):51–70, 1988.

[58] Umeshwar Dayal, Alejandro Buchmann, and Dennis R. McCarthy. Rules are objects too: A knowledge model for an active, object-oriented databasesystem. In *2nd International Workshop on Object-Oriented Database Systems (OODBS)*, Germany, 1988.

[59] Ahmed Elfatatry. Dealing with change: components versus services. *Communications of the ACM*, 50(8):35–39, 2007.

[60] Wilco Engelsman, Henk Jonkers, Henry M. Franken, and Maria-Eugenia Iacob. Architecture-driven requirements engineering. In Erik Proper, Frank Harmsen, and Jan L. G. Dietz, editors, *Advances in Enterprise Engineering II*, volume 28 of *Lecture Notes in Business Information Processing*, pages 134–154. Springer, 2009.

[61] Antonio Estruch and José Antonio Heredia Álvaro. Event-driven manufacturing process management approach. In *10th International Conference on Business Process Management (BPM)*, Estonia, 2012.

[62] Opher Etzion and Peter Niblett. *Event processing in action*. Manning Publications Co., 2010.

[63] Patrick Eugster. Type-based publish/subscribe: Concepts and experiences. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(1), 2007.

[64] Patrick Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[65] Patrick Eugster, Tobias Freudenreich, Sebastian Frischbier, Stefan Appel, and Alejandro Buchmann. Sound transformations for message passing systems. Technical report, Databases and Distributed Systems Group, TU Darmstadt, 2012.

[66] Patrick Eugster and K. R. Jayaram. EventJava: An extension of Java for event correlation. In *23rd European Conference on Object-Oriented Programming (ECOOP)*, Italy, 2009.

[67] Françoise Fabret, H. Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *ACM International Conference on Management of Data (SIGMOD)*, USA, 2001.

[68] Cao Fengyun and J. P. Singh. Efficient event routing in content-based publish-subscribe service networks. In *23rd International Conference of the IEEE Computer and Communications Societies (INFOCOM)*, China, 2004.

[69] Georg Feuerlicht. *Next Generation SOA: Can SOA Survive Cloud Computing?*, volume 67 of *Advances in Intelligent and Soft Computing*. Springer, 2010.

[70] Georg Feuerlicht and Shyam Govardhan. SOA: trends and directions. In *International Conference on Systems Integration (SI)*, Czech Republic, 2009.

[71] E. Fidler, Hans-Arno Jacobsen, Guoli Li, and Serge Mankovski. The PADRES distributed publish/subscribe system. In *Feature Interactions in Telecommunications and Software Systems VIII (FIW)*, UK, 2005.

[72] Ludger Fiege, Mariano Cilia, Gero Mühl, and Alejandro Buchmann. Publish/subscribe grows up: Support for management, visibility control & heterogeneity. *IEEE Internet Computing (Special Issue on Asynchronous Middleware and Services)*, 10(1), 2006.

[73] Tobias Freudenreich, Stefan Appel, Sebastian Frischbier, and Alejandro Buchmann. ACTrESS - automatic context transformation in event-based software systems. In *6th ACM International Conference on Distributed Event-Based Systems (DEBS)*, Germany, 2012.

[74] Tobias Freudenreich, Patrick Eugster, Sebastian Frischbier, Stefan Appel, and Alejandro Buchmann. Implementing federated object systems. In *27th European Conference on Object-Oriented Programming (ECOOP)*, France, 2013.

[75] Sebastian Frischbier, Alejandro Buchmann, and Dieter Pütz. FIT for SOA? Introducing the F.I.T. - Metric to optimize the availability of service oriented architectures. In *2nd International Conference on Complex Systems Design and Management (CSDM)*, France, 2011.

[76] Sebastian Frischbier, Michael Gesmann, Dirk Mayer, Andreas Roth, and Christian Webel. Emergence as competitive advantage - engineering tomorrow's enterprise software systems. In *14th International Conference on Enterprise Information Systems (ICEIS)*, Poland, 2012.

[77] Sebastian Frischbier, Alessandro Margara, Tobias Freudenreich, Patrick Eugster, David Eyers, and Peter Pietzuch. Aggregation for implicit invocations. In *12th International Conference on Aspect-Oriented Software Development (AOSD)*, Japan, 2013.

[78] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *7th European Conference on Object-Oriented Programming (ECOOP)*, Germany, 1993.

[79] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: why reuse is so hard. *IEEE Software Journal*, 12(6):17–26, 1995.

[80] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *4th International Symposium of VDM Europe on Formal Software Development*, The Netherlands, 1991.

[81] Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. EScala: modular event-driven object interactions in scala. In *10th ACM International Conference on Aspect-oriented Software Development (AOSD)*, Brazil, 2011.

[82] Kurt Geihs. Middleware challenges ahead. *IEEE Computer Journal*, 34(6):24–31, 2001.

[83] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003.

[84] Dominique Guinard, Vlad Trifa, Patrik Spiess, Bettina Dober, and Stamatis Karnouskos. SOA-based integration of the internet of things in enterprise services. In *IEEE International Conference on Web Services (ICWS)*, USA, 2009.

[85] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Elsevier Journal: Theoretical Computer Science*, 410(2–3):202–220, 2009.

[86] Wendi B. Heinzelman, Amy L. Murphy, Hervaldo S. Carvalho, and Mark A. Perillo. Middleware to support sensor network applications. *IEEE Network Journal*, 18(1):6–14, 2004.

[87] Nico Herzberg, Andreas Meyer, Oleh Khovalko, and Mathias Weske. Improving business process intelligence with object state transition events. In *32nd International Conference on Conceptual Modeling (ER)*. China, 2013.

[88] Nico Herzberg, Andreas Meyer, and Mathias Weske. An event processing platform for business process management. In *17th International Enterprise Distributed Object Computing Conference (EDOC)*, Canada, 2013.

[89] Annika Hinze, Kai Sachs, and Alejandro Buchmann. Event-based applications and enabling technologies. In *3rd ACM International Conference on Distributed Event-Based Systems (DEBS)*, USA, 2009.

[90] Volker Hoyer, Eva Bucherer, and Florian Schnabel. Collaborative e-business process modelling: Transforming private EPC to public BPMN business process models. In *5th International Conference on Business Process Management (BPM) Workshops*, Australia, 2007.

[91] Yi Huang and Dennis Gannon. A comparative study of web services-based event notification specifications. In *35th International Conference on Parallel Processing (ICPP) Workshops*, USA, 2006.

[92] Richard Hull, Elio Damaggio, Fabiana Fournier, Manmohan Gupta, Fenno (Terry) Heath III, Stacy Hobson, Mark Linehan, Sridhar Maradugu, Anil Nigam, Piyawadee Sukaviriya, and Roman Vaculin. Introducing the guard-stage-milestone approach for specifying business entity lifecycles. In *7th International Workshop on Web Services and Formal Methods (WS-FM)*. USA, 2010.

[93] Daniel Ingalls. The Smalltalk-76 programming system design and implementation. In *5th ACM Symposium on Principles of Programming Languages (POPL)*, USA, 1978.

[94] Ivan J. Jureta, Stéphane Faulkner, and Philippe Thiran. Dynamic requirements specification for adaptable and open service-oriented systems. In *5th International Conference Service-Oriented Computing (ICSOC)*, Austria, 2007.

[95] Matjaz B. Juric. WSDL and BPEL extensions for event driven architecture. *Elsevier Journal: Information and Software Technology*, 52(10):1023–1043, 2010.

[96] Gerhard Keller, August-Wilhelm Scheer, and Markus Nüttgens. *Semantische Prozeßmodellierung auf der Grundlage "Ereignisgesteuerter Prozeßketten (EPK)"*. Institut für Wirtschaftsinformatik, Saarbrücken, 1992.

[97] Rania Khalaf, Dimka Karastoyanova, and Frank Leymann. Pluggable framework for enabling the execution of extended BPEL behavior. In *5th International Conference Service-Oriented Computing (ICSOC) Workshops*, Austria, 2007.

[98] Pascal Kleber. Integration of event streams into EPCs and Software AG's model-to-execute process. Bachelor thesis, TU Darmstadt, 2013.

[99] Joshua Kramer. Advanced message queuing protocol (AMQP). *Linux Journal*, (187):3, 2009.

[100] Julian Krumeich, Benjamin Weis, Dirk Werth, and Peter Loos. Event-driven business process management: Where are we now? - a comprehensive synthesis and analysis of literature. *Emerald Business Process Management Journal (BPMJ)*, 20(4), 2014.

[101] Steffen Kunz, Tobias Fickinger, Johannes Prescher, and Klaus Spengler. Managing complex event processes with business process modeling notation. In *2nd International Workshop on Business Process Modeling Notation (BPMN)*, Germany, 2010.

[102] Zakir Laliwala and Sanjay Chaudhary. Event-driven service-oriented architecture. In *5th International Conference on Service Systems and Service Management (ICSSSM)*, Australia, 2008.

[103] Danny B. Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, 1999.

[104] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Transactions on Software Engineering (TSE)*, 33(10):709–724, 2007.

[105] Olga Levina and Vladimir Stantchev. Realizing event-driven SOA. In *4th International Conference on Internet and Web Applications and Services (ICIW)*. Italy, 2009.

[106] Christoph Liebig and Stefan Tai. Middleware mediated transactions. In *3rd International Symposium on Distributed Objects and Applications (DOA)*, Italy, 2001.

[107] Andoni Lombide Carreton, Stijn Mostinckx, Tom Cutsem, and Wolfgang Meuter. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In *48th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*. Spain, 2010.

[108] David Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman, 2002.

[109] David Luckham. *Event processing for business: organizing the real-time enterprise*. Wiley, 2011.

[110] Shruti P. Mahambre, S. D. Madhu Kumar, and Umesh Bellur. A taxonomy of QoS-aware, adaptive event-dissemination middleware. *IEEE Internet Computing Journal*, 11:35–44, 2007.

[111] N. Maiden. What has requirements research ever done for us? (goal-modeling techniques). *IEEE Software Journal*, 22(4):104–105, 2005.

[112] James McGovern, Oliver Sims, Ashish Jain, and Mark Little. *Enterprise service oriented architectures: concepts, challenges, recommendations*. Springer, 2006.

[113] René Meier and Vinny Cahill. Taxonomy of distributed event-based programming systems. In *22nd International Conference on Distributed Computing Systems (ICDCS) Workshops*, Austria, 2002.

[114] Marcelo R.N. Mendes, Pedro Bizarro, and Paulo Marques. Towards a standard event processing benchmark. In *4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, Czech Republic, 2013.

[115] J. Mendling, H.A. Reijers, and W.M.P. van der Aalst. Seven process modeling guidelines (7PMG). *Elsevier Journal: Information and Software Technology*, 52(2):127–136, 2010.

[116] Andreas Meyer, Sergey Smirnov, and Mathias Weske. Data in business processes. *EMISA Forum*, 31(3):5–31, 2011.

[117] Brenda M. Michelson. Event-driven architecture overview. *Patricia Seybold Group*, 2006.

[118] Joaquin Miller and Jishnu Mukerji. *MDA Guide, Version 1.0.1*. Object Management Group (OMG), 2003.

[119] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. Springer, 2006.

[120] Mohamed Nabeel, Stefan Appel, Elisa Bertino, and Alejandro Buchmann. Privacy preserving context aware publish subscribe systems. Technical Report CERIAS TR 2013-01, Center for Education and Research in Information Assurance and Security (CERIAS), Purdue University, 2013.

[121] Mohamed Nabeel, Stefan Appel, Elisa Bertino, and Alejandro Buchmann. Privacy preserving context aware publish subscribe systems. In *7th International Conference on Network and System Security (NSS)*, Spain, 2013.

[122] Mohamed Nabeel and Elisa Bertino. Towards attribute based group key management. In *18th ACM International Conference on Computer and Communications Security (CCS)*, USA, 2011.

[123] Mohamed Nabeel, Ning Shang, and Elisa Bertino. Efficient privacy preserving content based publish subscribe systems. In *17th ACM Symposium on Access Control Models and Technologies (SACMAT)*, USA, 2012.

[124] Mohamed Nabeel, Ning Shang, and Elisa Bertino. Privacy preserving policy-based content sharing in public clouds. *IEEE Transactions on Knowledge and Data Engineering*, 25(11):2602–2614, 2013.

[125] Stefan Nastic, Sanjin Sehic, Michael Vögler, Hong-Linh Truong, and Schahram Dustdar. PaTRICIA - a novel programming model for iot applications on cloud platforms. In *6th IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, Hawaii, 2013.

[126] Peter Niblett and Steve Graham. Events and service-oriented architecture: The OASIS web services notification specification. *IBM Systems Journal*, 44(4):869–886, 2005.

[127] OASIS Web Services Business Process Execution Language (WSBPEL) TC. *Web Services Business Process Execution Language (BPEL), Version 2.0*. 2007.

[128] Object Management Group (OMG). *Data Distribution Service (DDS) Specification, Version 1.2*. 2007.

[129] Object Management Group (OMG). *Business Process Model and Notation 2.0 by example*. 2010.

[130] Object Management Group (OMG). *Business Process Model and Notation (BPMN), Version 2.0*. 2011.

[131] John O'Hara. Toward a commodity enterprise middleware. *ACM Queue*, 5(4):48–55, 2007.

[132] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The information bus: an architecture for extensible distributed systems. *ACM SIGOPS Operating Systems Review*, 27(5):58–68, 1993.

[133] Oracle America, Inc. *JSR 342: Java Platform, Enterprise Edition 7 (Java EE 7) Specification*. 2013.

[134] Chun Ouyang, Marlon Dumas, Wil M.P. van der Aalst, Arthur ter Hofstede, and Jan Mendling. From business process models to process-oriented software systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 19(1):2:1–2:37, 2009.

[135] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT)*, Czech Republic, 1999.

[136] Michael P. Papazoglou, Paolo Traverso, Sharam Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *IEEE Computer Journal*, 40(11):38–45, 2007.

[137] Mike P. Papazoglou. Service-oriented computing: concepts, characteristics and directions. In *4th International Conference on Web Information Systems Engineering (WISE)*, Italy, 2003.

[138] Norman Paton. *Active rules in database systems*. Monographs in Computer Science. Springer, 1999.

[139] Norman W. Paton and Oscar Díaz. Active database systems. *ACM Computing Surveys (CSUR)*, 31(1):63–103, 1999.

[140] Peter Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In *22nd International Conference on Distributed Computing Systems (ICDCS) Workshops*. Austria, 2002.

[141] Peter Pietzuch, David Eyers, Samuel Kounev, and Brian Shand. Towards a common API for publish/subscribe. In *1st International Conference on Distributed Event-Based Systems (DEBS)*, Canada, 2007.

[142] Peter Pietzuch, Brian Shand, and Jean Bacon. A framework for event composition in distributed systems. In *4th ACM/IFIP/USENIX International Middleware Conference (Middleware)*, Brazil, 2003.

[143] Jeffrey S. Poulin. Measuring software reusability. In *3rd International Conference on Software Reuse (ICSR)*, Brazil, 1994.

[144] Costin Raiciu and David S. Rosenblum. Enabling confidentiality in content-based publish/-subscribe infrastructures. In *2nd International Conference on Security and Privacy in Communication Networks and the Workshops (SecureComm)*, USA, 2006.

[145] Rita A. Ribeiro, Ana M. Moreira, Pim van den Broek, and Afonso Pimentel. Hybrid assessment method for software engineering decisions. *Elsevier Journal: Decision Support Systems*, 51(1):208–219, 2011.

[146] David S. Rosenblum and Alexander L. Wolf. A design framework for internet-scale event observation and notification. *ACM SIGSOFT Software Engineering Notes*, 22(6):344–360, 1997.

[147] Douglas T. Ross and Kenneth E. Schoman Jr. Structured analysis for requirements definition. *IEEE Transactions on Software Engineering (TSE)*, 3(1):6–15, 1977.

[148] Szabolcs Rozsnyai, Josef Schiefer, and Alexander Schatten. Concepts and models for typing events for event-based systems. In *1st International Conference on Distributed Event-Based Systems (DEBS)*, Canada, 2007.

[149] Kai Sachs. *Performance modeling and benchmarking of event-based systems*. PhD thesis, TU Darmstadt, Germany, 2011.

[150] Kai Sachs, Stefan Appel, Samuel Kounev, and Alejandro Buchmann. Benchmarking publish/subscribe-based messaging systems. In *15th International Conference on Database Systems for Advanced Applications (DASFAA) Workshops*, Japan, 2010.

[151] Kai Sachs, Samuel Kounev, Stefan Appel, and Alejandro Buchmann. Benchmarking of message-oriented middleware. In *3rd ACM International Conference on Distributed Event-Based Systems (DEBS)*, USA, 2009.

[152] Kai Sachs, Samuel Kounev, Stefan Appel, and Alejandro Buchmann. A performance test harness for publish/subscribe middleware. In *ACM SIGMETRICS/Performance Demo Competition*, USA, 2009.

[153] Kai Sachs, Samuel Kounev, Jean Bacon, and Alejandro Buchmann. Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark. *Elsevier Journal: Performance Evaluation*, 66(8):410–434, 2009.

[154] Guido Salvaneschi, Joscha Drechsler, and Mira Mezini. Towards distributed reactive programming. In *15th International Conference on Coordination Models and Languages (COORDINATION)*, Italy, 2013.

[155] August-Wilhelm Scheer and Markus Nüttgens. ARIS architecture and reference models for business process management. In Wil Aalst, Jörg Desel, and Andreas Oberweis, editors, *Business Process Management*, volume 1806 of *Lecture Notes in Computer Science*, pages 376–389. Springer, 2000.

[156] Markus Schief, Christian Kuhn, Birgit Zimmermann, Philipp Rösch, Walter Waterfeld, Jens Schimmelpfennig, Dirk Mayer, Heiko Maus, and Jörn Eichler. The ADiWa project - on the way to just-in-time process dynamics based on events from the internet of things. In *13th International Conference on Enterprise Information Systems (ICEIS)*, China, 2011.

[157] Christian Seeger, Alejandro Buchmann, and Kristof Van Laerhoven. myHealthAssistant: A phone-based body sensor network that captures the wearer's exercises throughout the day. In *6th International Conference on Body Area Networks (BodyNets)*, China, 2011.

[158] Christian Seeger, Alejandro Buchmann, and Kristof Van Laerhoven. An event-based BSN middleware that supports seamless switching between sensor configurations. In *ACM International Health Informatics Symposium (IHI)*, USA, 2012.

[159] Ning Shang, Mohamed Nabeel, Federica Paci, and Elisa Bertino. A privacy-preserving approach to policy-based content dissemination. In *26th International Conference on Data Engineering (ICDE)*, USA, 2010.

[160] Mudhakar Srivatsa and Ling Liu. Securing publish-subscribe overlay services with eventguard. In *12th ACM International Conference on Computer and Communications Security (CCS)*, USA, 2005.

[161] Mudhakar Srivatsa and Ling Liu. Secure event dissemination in publish-subscribe networks. In *27th International Conference on Distributed Computing Systems (ICDCS)*, Canada, 2007.

[162] Sebastian Stein. *Modelling Method Extension for Service-Oriented Business Process Management*. PhD thesis, Christian-Albrechts-Universität zu Kiel, Germany, 2009.

[163] STOMP Working Group. *STOMP Protocol Specification, Version 1.2*. 2012.

[164] Heiner Stuckenschmidt and Holger Wache. Context modeling and transformation for semantic interoperability. In *7th International Workshop on Knowledge Representation meets Databases (KRDB)*, Germany, 2000.

[165] Florin Sultan, Aniruddha Bohra, and Liviu Iftode. Service continuations: an operating system mechanism for dynamic migration of internet service sessions. In *22nd Symposium on Reliable Distributed Systems (SRDS)*, Italy, 2003.

[166] Sun Microsystems, Inc. *Java Message Service (JMS) Specification, Version 1.1*. 2002.

[167] C. Timurhan Sungar, Patrik Spiess, Nina Oertel, and Oliver Kopp. Extending BPMN for wireless sensor networks. In *15th IEEE Conference on Business Informatics (CBI)*, Austria, 2013.

[168] Stefan Tai, Philipp Leitner, and Sharam Dustdar. Design by units: Abstractions for human and compute resources for elastic systems. *IEEE Internet Computing*, 16(4):84 –88, 2012.

[169] Hugh Taylor, Angela Yochem, Les Phillips, and Frank Martinez. *Event-driven architecture: How SOA enables the real-time enterprise*. Addison-Wesley, 2009.

[170] Wil M. P. van der Aalst. Formalization and verification of event-driven process chains. *Elsevier Journal: Information and Software Technology*, 41(10):639–650, 1999.

[171] Axel van Lamsweerde. Goal-oriented requirements engineering: a guided tour. In *5th IEEE International Symposium on Requirements Engineering (RE)*, Canada, 2001.

[172] Steve Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, 10(6):87–89, 2006.

[173] Holger Wache and Heiner Stuckenschmidt. Practical context transformation for information system interoperability. In *3rd International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT)*, UK, 2001.

[174] Chenxi Wang, Antonio Carzaniga, David Evans, and Alexander L. Wolf. Security issues and requirements for internet-scale publish-subscribe systems. In *35th Hawaii International Conference on System Sciences (HICSS)*, Hawaii, 2002.

[175] Matthias Weidlich, Holger Ziekow, Jan Mendling, Oliver Günther, Mathias Weske, and Nirmit Desai. Event-based monitoring of process execution violations. In *9th International Conference on Business Process Management (BPM)*, France, 2011.

[176] Rainer Weinreich and Johannes Sametinger. *Component-Based Software Engineering: Putting the Pieces Together*, chapter Component models and component services: Concepts and principles. Prentice Hall, 2001.

[177] Jennifer Widom, Stefano Ceri, and Umeshwar Dayal. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers Inc., 1995.

[178] Matthias Wieland, Daniel Martin, Oliver Kopp, and Frank Leymann. SOEDA: A method for specification and implementation of applications on a service-oriented event-driven architecture. In *12th International Conference on Business Information Systems (BIS)*, Poland, 2009.

[179] Workflow Management Coalition. *Workflow Management Coalition Workflow Standard: Process Definition Interface–XML Process Definition Language (XPDL)*. 2012.

[180] World Wide Web Consortium (W3C). *Web Services Eventing (WS-Eventing)*. 2006.

[181] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *ACM International Conference on Management of Data (SIGMOD)*, USA, 2006.

[182] Sebastian Zöller, Markus Wachtel, Fabian Knapp, and Ralf Steinmetz. Going all the way - detecting and transmitting events with wireless sensor networks in logistics. In *38th International Conference on Local Computer Networks (LCN) Workshops*, Australia, 2013.