

Simplifying the Use of Event-based Systems With Context Mediation and Declarative Descriptions

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte
Dissertation zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
von M.Sc. Tobias Freudenreich aus Fulda (Hessen)

Tag der Einreichung: 11. Mai 2015, Tag der Prüfung: 26. Juni 2015
Darmstadt 2015 — D 17

1. Gutachten: Prof. Alejandro Buchmann, Ph.D.
2. Gutachten: Prof. Patrick Eugster, Ph.D.
3. Gutachten: David Eysers, Ph.D.



TECHNISCHE
UNIVERSITÄT
DARMSTADT



DVS

Bitte zitieren Sie dieses Dokument als:

URN: urn:nbn:de:tuda-tuprints-51319

URL: <http://tuprints.ulb.tu-darmstadt.de/5131>

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

<https://creativecommons.org/licenses/by-nc-nd/3.0/de/>

Curriculum Vitae

TOBIAS FREUDENREICH

Personal Information

Date of Birth July 2nd, 1984
Place of Birth Fulda (Hessen), Germany

Education

2011 – 2015
[Technische Universität Darmstadt](#)
PhD Student

2009 – 2011
[Technische Universität Darmstadt](#)
M.Sc. Computer Science

2008 – 2009
[Carnegie Mellon University](#)
Human-Computer-Interaction

2005 – 2008
[Technische Universität Darmstadt](#)
B.Sc. Computer Science

2004
[Freiherr-vom-Stein-Schule Fulda](#)
Abitur

Positions

2011 – 2015
[Technische Universität Darmstadt](#)
Research and Teaching Assistant

Awards

2007
[Fulbright Scholarship](#)




Zusammenfassung

Das Internet der Dinge und das damit verbundene stark zunehmende Aufkommen von Sensoren sind Trends, die zu Cyber-physical Systems führen. Eine charakteristische Eigenschaft dieser Systeme ist, dass viele verschiedene Komponenten durch Austausch von Ereignissen kommunizieren. Diese Abstraktion ist sehr nützlich, um das hohe Datenaufkommen dieser Systeme verarbeiten zu können. Allerdings sind Cyber-physical Systems dadurch auch sehr komplex und nur speziell ausgebildete Informatiker können diese korrekt konfigurieren und verwenden.

Einer der Hauptgründe für diese Komplexität ist fehlende Kontextinformation in den Ereignissen. In dieser Arbeit haben wir deshalb den Kontext von Ereignissen untersucht und haben zwei unterschiedliche Dimensionen erkannt: Kontext der Ereignisdaten und Kontext des Ereignisses selbst. Kontext der Ereignisdaten sind Annahmen die zur Interpretation der Daten beitragen, wie z.B. das verwendete Einheitensystem oder die Struktur der Daten. Kontext des Ereignisses selbst umfasst Informationen, die zum Ereignis gehören, in diesem aber nicht enthalten sind. Bei einem Ereignis, welches Positionsdaten enthält, kann dies beispielsweise der zugehörige Raum sein.

Kontext der Ereignisdaten hilft die Heterogenität von Cyber-physical Systems zu überwinden. Ereignisproduzenten und -konsumenten machen oft unterschiedliche Annahmen bei der Interpretation von Ereignisdaten, was zu unterschiedlichen Interpretationen des gleichen Ereignisses führen kann. Um dies zu beheben, haben wir ACTrESS entwickelt. ACTrESS bietet eine Möglichkeit die Interpretationsannahmen explizit in einem sogenannten *interpretation context* zu kodieren. Dieser wird dann zur Middleware gesendet, welche damit Ereignisse zwischen unterschiedlichen Kontexten durch Transformation übersetzen kann. ACTrESS analysiert dabei die gesendeten Kontexte und generiert Transformatoren, welche dynamisch nachgeladen werden. Dadurch vermeiden wir teure Mechanismen, wie z.B. Reflection. Die Evaluation von ACTrESS bestätigt das: die Performanz lässt sich nicht von der eines reinen content-based Publish/Subscribe Systems unterscheiden.

Da Ereignisse keine kontextuellen Informationen mitbringen, müssen komplexe Situationen von sehr erfahrenen Informatikern mit Hilfe von speziellen Regeln beschrieben werden. Dabei werden oft mehrere Ereignisse miteinander kombiniert. Allerdings versprechen Cyber-physical Systems unser alltägliches Leben zu verändern, was z.B. bei sogenannten intelligenten Häusern schon heute erkennbar ist. Die oben aufgezeigte Komplexität verhindert aber, dass die eigentlichen Nutzer die Systeme entsprechend nutzen können. Deshalb haben wir in dieser Arbeit eine deklarative Sprache entwickelt, mit der komplexe Situationen und entsprechende Reaktionen darauf leicht beschrieben werden können. Dazu beschreibt die Arbeit einen Mechanismus, der diese Beschreibung in ausführbaren Code umwandelt. Die Kernidee dabei ist, dass Ereignisse kontextualisiert werden. Unsere Middleware reichert eingehende Ereignisse basierend auf der Situationsbeschreibung mit fehlender kontextueller Information an. Die so angereicherten Ereignisse werden entsprechend kombiniert und korreliert, um schlussendlich die beschriebene Situation detektieren zu können. Wir generieren kleine Berechnungseinheiten, wodurch eine gute Parallelisierung erreicht werden kann. Ebenso kann man so sehr einfach hoch- und runterskalieren, wodurch unsere Middleware sehr gut zu modernen Cloudarchitekturen passt. Wir haben unseren Ansatz mit einer Usability Studie, sowie Performanzmessungen untersucht. Die Studie zeigt, dass unsere Sprache das Beschreiben reaktiven Verhaltens in Cyber-physical Systems signifikant vereinfacht. Die Performanzmessungen zeigen, dass die Möglichkeit der automatischen Skalierung und Verteilung etwas Leistung im



Vergleich zu Esper kostet. Esper ist dabei eine hoch-optimierte Software, die jedoch die angesprochenen Vorteile nicht bietet.

Abstract

Current trends like the proliferation of sensors or the Internet of Things lead to Cyber-physical Systems (CPSs). In these systems many different components communicate by exchanging events. While events provide a convenient abstraction for handling the high load these systems generate, CPSs are very complex and require expert computer scientists to handle correctly.

We realized that one of the primary reasons for this inherent complexity is that events do not carry context. We analyzed the context of events and realized that there are two dimensions: context about the data of an event and context about the event itself. Context about the data includes assumptions like systems of measurement units or the structure of the encoded information that are required to correctly understand the event. Context about the event itself is data that provides additional information to the information carried by the event. For example an event might carry positional data, the additional information could then be the room identifier belonging to this position.

Context about the data helps bridge the heterogeneity that CPSs possess. Event producers and consumers may have different assumptions about the data and thus interpret events in different ways. To overcome this gap, we developed the ACTrESS middleware. ACTrESS provides a model to encode interpretation assumptions in an *interpretation context*. Clients can thus make their assumptions explicit and send them to the middleware, which is then able to mediate between different contexts by transforming events. Through analysis of the provided contexts, ACTrESS can generate transformers, which are dynamically loaded into the system. It does not need to rely on costly operations like reflection. To prove this, we conducted a performance study which shows that in a content-based publish/subscribe system, the overhead introduced by ACTrESS' transformations is too small to be measurable.

Because events do not carry contextual information, expert computer scientists are required to describe situations that are made up of multiple events. The fact that CPSs promise to transform our everyday life (e.g., smart homes) makes this problem even more severe in that most of the target users cannot use CPSs. In this thesis, we developed a declarative language to easily describe situations and a desired reaction. Furthermore, we provide a mechanism to translate this high-level description to executable code. The key idea is that events are contextualized, i.e. our middleware enriches the event with the missing contextual information based on the situation description. The enriched events are then correlated and combined automatically, to ultimately be able to decide if the described situation is fulfilled or not. By generating small computational units, we achieve good parallelization and are able to elegantly scale up and down, which makes our approach particularly suitable for modern cloud architectures. We conducted a usability analysis and performance study. The usability analysis shows that our approach significantly simplifies the definition of reactive behavior in CPS. The performance study shows that the achieved automatic distribution and parallelization incur a small performance cost compared to highly optimized systems like Esper.



Acknowledgements

There are a number of people that helped me on my path to a PhD. I am eternally grateful for their support which I was lucky to receive. I doubt that without this support I could have completed this thesis.

I want to express my gratitude towards my advisor Professor Alejandro Buchmann, who did not only provide continuous guidance and advice for my thesis, but also provided emotional support. I wish to express special thanks to Professor Patrick Eugster. Collaborating with him shone light from different angles on our approaches and led to many fruitful discussions which helped strengthen our approaches. Also, he was kind enough to review my thesis. I want to thank Senior Lecturer David Eyers for reviewing my thesis, the nice discussions and hands-on help. Furthermore, I wish to say thank you to Prof. Stefan Roth, Prof. Mira Mezini and Prof. Karsten Weihe for taking time to be on my dissertation defense committee.

I specifically want to thank Professor Matthias Hollick, who motivated me to pursue a PhD.

My gratitude also extends to my colleagues of Databases and Distributed Systems group at Darmstadt. Through countless discussions, they helped sharpen my ideas and provided continuous feedback on solutions. I want to specifically thank Stefan Appel and Sebastian Frischbier for collaborating with me in various research and teaching activities. I also wish to thank Maria Tiedemann for her wonderful and competent support in the many administrative activities.

I also want to thank my students Henriette Röger who tackled the vast amount of EPCs in Software AG's Industry Performance Ready database, and Prashanth Sadanand who dug into my code and implemented a socket-based communication.

Software AG was kind enough to provide their Industry Performance Ready database for analytical studies, which helped verifying my assumptions. Besides them, I also want to thank SoSci Survey, a powerful online survey platform for providing a free-of-charge survey project.

Finally, I wish to deeply thank my family and friends. My parents deserve special thanks for supporting me on the path I chose, always motivating, but never pushing. The magnitude of such support is impossible to express with words. And to my friends: You made me laugh, have a good time, you provided emotional and mental support, you provided support beyond anything that words can express. Without you, I would not be who I am.



Contents

I. Introduction and Background	1
1. Introduction	3
1.1. Problem Statement	4
1.2. Motivating Scenario	5
1.3. Contributions	6
1.3.1. ACTrESS: Handling Heterogeneity	6
1.3.2. Policy-driven Contextualization: Targeted Event Enrichment	7
1.4. Related Activities and Publications	8
1.4.1. Research Projects	8
1.4.2. Research Collaborations	8
1.4.3. Related Publications	8
1.5. Structure	9
2. Background	11
2.1. Publish/Subscribe Systems	11
2.2. Event-based Systems	13
2.2.1. Complex Event Processing	14
2.2.2. Cyber-physical Systems	16
3. Context in Event-based Systems	17
3.1. Context Definition	17
3.2. Context Model	17
3.2.1. Context About Data: Interpretation Context	18
3.2.2. Context About Events: Situation Context	19
II. Concepts and Implementation	27
4. ACTrESS	29
4.1. Transformation Framework	31
4.1.1. Event Representation	31
4.1.2. Transformation Rules	32
4.1.3. Transformation Space	36
4.2. Interpretation Contexts	38
4.2.1. Interpretation Context Specialization	40
4.2.2. Augmenting Interpretation Contexts	42
4.2.3. Modifying Existing Interpretation Contexts	42
4.3. ACTrESS Architecture and Implementation	43
4.3.1. Architecture	43
4.3.2. Declaring Interpretation Context	45
4.3.3. Transformation in Action	46

4.4. Evaluation	50
4.4.1. Performance Evaluation	50
4.4.2. Ease of Use	57
5. Policy-driven Contextualization	61
5.1. Ontology Model	62
5.1.1. Model Choice	63
5.1.2. Validation Against Industry-ready Event-driven Process Chains (EPCs)	66
5.2. Policy Language	66
5.3. Contextualization Mechanisms: From Declarative Policy Language (DPL) to Executable Code	68
5.3.1. Setting the Stage: The Startup Phase	68
5.3.2. Setting Up: Policy Registration	70
5.3.3. Setting Things in Motion: Code Generation	73
5.3.4. Settling: Policy Shutdown	83
5.4. Architecture	83
5.4.1. Event and Command Bus	83
5.4.2. User Interface	83
5.4.3. Policy Engine	85
5.4.4. ACTrESS	85
5.4.5. Worker Nodes	86
5.5. Implementation	87
5.6. Evaluation	89
5.6.1. Scenarios	89
5.6.2. Cognitive Dimensions	91
5.6.3. User Study: Understandability of Code	97
5.6.4. Performance Evaluation	99
5.6.5. Summary of the Evaluation	106
III. Related Work and Conclusion	107
6. Related Work	109
6.1. ACTrESS	109
6.2. Contextualization	115
7. Conclusion and Future Work	123
7.1. Future Work	125
7.1.1. ACTrESS	125
7.1.2. Policy-driven Contextualization	125

List of Figures

2.1. General architecture of publish/subscribe systems	11
2.2. Illustration of event creation	13
2.3. Example of Complex Event Processing (CEP)	15
3.1. Overview of contexts surrounding an event	18
3.2. Different event types for the same semantics	19
4.1. Architectural overview of ACTrESS	30
4.2. Example event type of a invoice line	32
4.3. Illustration of the transformation space granularity	37
4.4. Interpretation context example	39
4.5. Example root interpretation context	39
4.6. US interpretation context	41
4.7. Sample transformation	41
4.8. Interpretation context specialization and resulting transformation paths.	42
4.9. Abstract architecture of our implementation.	43
4.10. Sequence diagram for the code generation.	48
4.11. Performance setup	51
4.12. Throughput/latency for the SPECjms2007-inspired workload.	53
4.13. Producer/Consumer ratio	54
4.14. Impact of Event Type Structure	55
4.15. Impact of Locality of the Transformation	55
4.16. Generation time	56
4.17. Recompile overhead	57
5.1. Policies are specified and mapped to executable code	62
5.2. Conceptual view of our ontology model	63
5.3. Example ontology to the policy in Listing 5.1	64
5.4. Example policy logic tree	73
5.5. Schematic of a leaf-Event Enrichment Component (EEC)	74
5.6. Schematic of a query-EEC	75
5.7. Schematic of a correlation-EEC	77
5.8. Illustration of the activation index	78
5.9. Example of a policy logic tree with a relationship node with zero event-based attributes	79
5.10. Schematic of a followed-by-EEC	80
5.11. Example why early deactivation does not work	81
5.12. Architecture for the DPL-enabled middleware	83
5.13. Screenshot of the policy editor and ontology model editor	84
5.14. Contextualization Sequence	88
5.15. DPL is not overly terse	93
5.16. Correctness results of the user study	98
5.17. Perceived easiness of labeling the statements	99
5.18. Average response time for answering the questions	100

5.19. Simulation to create the event sequence	101
5.20. Setup for the performance measurements	102
5.21. Utilization over time for 1300 events / second	103
5.22. CPU utilization of the Contextualizer	104
5.23. Standard deviation of the CPU utilization	104
5.24. CPU utilization of ActiveMQ	105
5.25. Achievable throughput	106
7.1. Illustration of the generic task model of Eventlets [8]	127

List of Tables

3.1. Context elements of various context-aware systems.	22
3.2. Requirements for context models	24
4.1. Overview of ACTrESS's Java annotations.	47
4.2. Code complexity comparison	59
5.1. Overview of followed-by conversions to reach Extended Conjunctive Normal Form (ECNF)	72
5.2. Example of the activation index	78
5.3. Illustration of activations and NOT-relationships interplay	82
5.4. Illustration of the match-finding algorithm	82
5.5. Summary of cognitive dimensions analysis	97
6.1. Feature comparison. ✓ refers to supported features, and ~ to weakly addressed require- ments.	110



List of Abbreviations

BPMN	Business Process Model and Notation
CEP	Complex Event Processing
CNF	Conjunctive Normal Form
CPS	Cyber-physical System
DAO	Data Access Object
DBMS	Database Management System
DPL	Declarative Policy Language
EAI	Enterprise Application Integration
EBNF	Extended Backus-Naur Form
EBS	Event-based System
ECA	Event-Condition-Action
ECNF	Extended Conjunctive Normal Form
EDA	Event-driven Architecture
EEC	Event Enrichment Component
EMF	Eclipse Modeling Framework
EPA	Event Processing Agent
EPTS	Event Processing Technical Society
EPC	Event-driven Process Chain
EPL	Event Processing Language
EQL	Event Query Language
ERM	Entity-Relationship Model
GMF	Graphical Modeling Framework
IoT	Internet of Things
JMS	Java Message Service
MAPE	Monitoring-Analysis-Planning-Execution
M2M	Machine-to-Machine
MOM	Message-oriented Middleware
NTP	Network Time Protocol
OODA	Observe-Orient-Decide-Act

OOP	Object-oriented Programming
OWL	Web Ontology Language
PDCA	Plan-Do-Check-Action
SOA	Service-oriented Architecture
SPU	Event Stream Processing Unit
WSN	Wireless sensor network
WSAN	Wireless sensor and actor network
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSLT	XSL Transformations

Part I.

Introduction and Background



1 Introduction

The human spirit must prevail over technology.

–Albert Einstein

Our world is becoming more digital every day. While in traditional computer systems, data was input by humans, the trend today is to let computer systems perceive the world through various sensors. A very simple, yet omnipresent example are mobile phones. Traditional mobile phones were mostly used for calling and writing short messages to others. With the dawn of the smartphones, mobile phones became sensor-rich, multi-purpose tools which are used for many things beyond calling.

We can see this proliferation of sensors in various domains: Modern smartphones carry multiple small sensors in their cases. The Android operating system knows 20 different kinds of sensors¹. Sensors revolutionize production factories. Companies outfit their production sites with a multitude of sensors to increase productivity of workers and to predict irregularities in order to minimize downtimes. For example, in 2013 General Electric built a new production plant with 10000 sensors on 16000 square meters [82]. Other companies experiment with *Smart Factories*, in which individual parts find their way through the production plant, instead of being guided externally. This trend is often summarized by the key term *Industry 4.0*. Logistics providers are able to offer new services to their customers. Detailed arrival time predictions and continuous good monitoring (e.g., temperature readings to prove an uninterrupted cold chain) are just two examples which make logistics providers use more and more sensors.

Smart homes and smart cars are further examples which show pervasiveness of sensors in our everyday lives. Smart homes aim at supporting people by doing various tasks automatically. Famous examples are the intelligent refrigerator or fall detection sensors for elderly people. Cars include more and more sensors as well. From monitoring critical motor parts to driver assistance functions like a distance sensor: sensors are omnipresent in cars.

This trend, that even subparts of the items we use have sensors and communicate with other parts, is called the *Internet of Things (IoT)* [12]. The physical and the virtual world merge into sense-and-respond systems, which observe the environment through various sensors and use actuators for carrying out actions in the physical world. These Cyber-physical Systems (CPSs) [184] are predicted to transform the way devices communicate with each other, similar to how the Internet transformed human interaction [157]

As with every new technology, CPSs bring new challenges. Communication in CPSs happens in a Machine-to-Machine (M2M) fashion. Components interact by exchanging events. Due to the large number of participating components (e.g., sensors), a vast amount of events must be processed. Processing them with traditional request/reply mechanisms, however, does not scale to the large number of sensors we observe today. To handle the load, systems must rely on a push-based interaction paradigm used in Event-based Systems (EBSs). In EBSs, sensor data is pushed from sensors to interested consumers as events. Components no longer actively query for data, but have to deal with what is pushed to them.

¹ <http://developer.android.com/reference/android/hardware/Sensor.html>

While EBSs are very apt to handling the performance requirements of CPSs, there are still unsolved challenges, in particular *heterogeneity* and *contextualization*. CPSs rarely have a central governing instance and integrate a multitude of different kinds of sensors, software components and other parts from various vendors. It is thus in their nature to be heterogeneous. This becomes apparent in today's economy, where large-scale software systems manage complex supply-chain networks, comprising different companies around the globe. Cooperation happens between companies of different countries, cultures and structures. For example, today's complex supply chains involve many companies world-wide; production strategies like *just-in-time production* have strongly increased the need for continuous low-latency flows of information between participants in the chain (e.g., monitoring transported goods) [35].

Thus, we face an increasingly heterogeneous software landscape, in which components are developed independently, and yet have to communicate in a scalable, flexible and low-latency manner. The problem that arises in such heterogeneous environments is different data representations and semantics. Local interpretations differ due to geographical, cultural, legal, and technical reasons. However, a correct interpretation is crucial to properly handle and interpret events.

As sense-and-respond systems, CPSs react to situations. However, a situation is a high-level abstraction of a certain combination of low-level events. It must thus be encoded in some way for machines to be able to detect it. Events must be properly contextualized with the target situation in mind. Today, Event-Condition-Action (ECA) rules are the method of choice. These declarative rules, however, have a very low abstraction level. They require expert developers to handle correctly. In addition, even when handled by experts, the number of rules quickly exceeds a manageable number, creating a maintenance nightmare with hidden dependencies among the rules and no indication why a rule was created in the first place.

The problem underlying both challenges is the fact that events do not yield contextual information, neither about their interpretation assumptions (e.g., the unit system) nor about their data (e.g., a room associated with the event's position information). This missing contextual information has to be manually added, which leads to increased implementation effort and complexity. Thus, CPSs require software engineers with special training. However, the use cases above illustrate that the real users are domain experts, not developers. The goal of this thesis is to provide an approach for using contextual information to tackle *heterogeneity* and *contextualization*.

1 Problem Statement

In recent years, event processing gained a lot of attention and rose from pure academic research to a field with a broad spectrum of industry products. Most famous are Complex Event Processing (CEP) engines and message-oriented middleware for event dissemination. However, despite all the effort by companies to increase the usability of their products with graphical interfaces, using these products remains a computer scientist's task. Support for domain experts is yet missing. In a recent survey Derler et al. looked at methods for developing CPSs and concluded that modeling and developing CPSs is very challenging [68]. Other researchers also identified the lack of appropriate mechanisms, especially regarding contextualization [179].

Examining the history of database systems showcases the demand for domain expert support. Database Management Systems (DBMSs) abstract from the actual way that data is stored and provide a unified way of accessing this data [57]. While in the beginning software engineers enjoyed the benefit of storing data without worrying about details of the underlying file system and being able to easily share data between applications, they soon realized that the query language (e.g., SQL) provided by the DBMS can

also be used by domain experts. Since users only declaratively *describe* which data they want to have, without worrying about access paths and other details, retrieving data became much easier.

When using EBSs, developers typically want the system to react to certain situations. The problem is that these situations are usually comparatively complex, while the available events are low-level observations. The challenge is to compose these low-level events until they are at the abstraction-level of the desired situation. This is similar to extracting and combining information scattered across structured files. Similar to the idea behind SQL, it would be desirable to let users of EBSs describe these situations in a declarative way and let the system figure out how to assign meaning to the events.

This requires assigning meaning to events by putting them into context. This entails the challenges of heterogeneity and contextualization. Thousands of sensors will produce events with different structure and value semantics. This must automatically be mediated. However, existing mediation approaches from databases cannot be applied due to the inherent anonymity between components. The main challenge of contextualizing events is choosing the right information to use for event enrichment. In fact, the recent survey "Context Aware Computing for The Internet of Things: A Survey" mandates that there is a lot of context data for enrichment, but what to enrich exactly is a huge challenge [152].

Research Questions

Summarizing, we have the following research questions:

- What role does contextual information play in EBSs?
- What is the interplay between EBSs and traditional context-aware systems?
- How can we support multiple interpretations of data in EBSs?
- What is a good model for the mediation of events?
- How can we allow for the higher-level definition of reactive behavior?
- How can we enrich events with the right external information?

1 Motivating Scenario

An example application that showcases the usefulness of our approach is from the domain of situation monitoring, originally proposed by Georgakopoulos et al. [98]: Among several security policies, a company has the following policy: generally, guests may walk around company facilities freely. However, some rooms contain company secrets or are otherwise restricted. In these rooms, guests need to be accompanied by an employee who must be within 5 meters of the guest, otherwise an alarm should sound. There are many sensors in place to derive positions of people, for example RFID readers and cameras. We assume that each of these sensors is able to determine the position of a person (relative to itself) and some kind of identity information of that person. For example, RFID readers can determine the position of smart badges and use the tag's id to uniquely identify the person by using a mapping from tag ids to person ids. Cameras might do face recognition and compare this information to image data, classifying any unknown person as a guest.

We want to showcase the challenge of heterogeneity by looking at position data in this scenario. RFID readers report the position of tracked tags in Cartesian coordinates relative to their position. However,

room coordinates are stored as company-wide Cartesian coordinates. Thus, events from RFID readers need coordinate transformation. Similarly, cameras are likely to report positions as relative, polar coordinates, which again must be transformed to Cartesian and then to company-wide coordinates.

Deriving possible security policy violations from the position data of employees and guests requires position events to be contextualized: the system must 1) query a database to determine if an observed person is known; 2) add/translate the reporting sensor's position information; 3) check this position against the list of restricted rooms; 4) correlate this information with other position events, to check if a guest is accompanied. Each of these steps require a number of ECA rules to encode, resulting in a more and more complex hierarchy of events to ultimately check the given policy. For example, querying a database for the status of a person typically involves a rule to represent the contents of the appropriate database table as a static stream. This stream must then be joined with the position stream and filtered for the desired status (e.g., employees). As we will see in Chapter 5, these rules are non-trivial to write and bear technical details. When adding more "real-world" requirements like different security clearances, the set of rules quickly becomes unmanageable.

1 Contributions

In this thesis, we introduce a middleware approach to handle event heterogeneity and event contextualization in EBS. Our approach provides users with a declarative language which they can use to intuitively specify the behavior of the underlying EBS. Our contributions can be grouped into two parts: heterogeneity handling and targeted event enrichment. In the heterogeneity part we introduce ACTrESS, our middleware approach for handling heterogeneity. Targeted event enrichment uses the policy-style behavior description to enrich events with the right data to execute desired behavior. We incorporate heterogeneity handling into this mechanism to obtain full contextualization support.

1.3.1 ACTrESS: Handling Heterogeneity

To handle heterogeneity in EBS, we developed ACTrESS (Automatic Context Transformations in Event-based Software Systems). ACTrESS has a theoretical foundation, which defines the expressive power of the language which defines the transformations. Furthermore, we implemented a prototype of this foundation based on industry-strength Message-oriented Middleware (MOM).

The contributions of the theoretical foundation are:

- We identify challenges for contextual transformation in EBSs, useful for classifying heterogeneity-mediating approaches.
- We model data interpretation contexts as modular, reusable units tailored to the characteristics of EBSs, providing easy maintainability and supporting different levels of flexibility.
- We present an expressive and extensible model for federated software systems, centered on transformation rules.
- We suggest a distributed architecture that allows for mediation and updates at runtime, supporting the dynamic nature of EBSs.

The contributions of the ACTrESS implementation are:

- We provide a prototype implementation of our transformation model as a plugin for the popular, open-source JMS broker ActiveMQ [189]. This demonstrates that the proposed principles can be integrated into industry-strength middleware.

-
- We show how our transformation model enables runtime code generation for transformation application.
 - We evaluate our prototype, showing that our solution does not impose any measurable overhead over content-based publish/subscribe and is superior to approaches employing techniques based on self-describing messages.
 - We also show that our approach reduces coding efforts over client-side transformations both for initial coding and upon code adaptation.

1.3.2 Policy-driven Contextualization: Targeted Event Enrichment

We suggest an approach which allows domain experts to specify desired behavior in a declarative way. We call these specifications *policies*, which we use to guide the event enrichment mechanism. Our approach is similar to query languages in DBMSs, in which a database expert creates the schema, allowing domain users to query the data: after an expert specifies the domain ontology, which serves as the schema, domain users can use a declarative language to specify behavior. The language elements are automatically adapted to the target domain.

The contributions of our contextualization approach are:

- We suggest a novel approach to handle the complexity of Cyber-physical Systems and EBSs in general. Our approach facilitates the use of EBSs by domain experts.
- We introduce policies as an intuitive facility to describe reactive behavior.
- We provide a method for normalizing boolean terms which include a sequence operator.
- We provide a generic, declarative language to state policies, which adapts to the target domain. The language can be mapped to multiple concrete languages and implementations.
- We suggest a generic middleware architecture for performing event contextualization.
- We have implemented our approach as a middleware, which relies on MOM. We used the popular, open-source JMS broker ActiveMQ [189].
- Our implementation generates many small, autonomous event processing components based on the policies.
- Our middleware can distribute these processing components on an arbitrary number of nodes, supporting scaling in modern cloud environments elegantly.
- Our approach preserves the benefits of Event-driven Architectures (EDAs): new sensors/producers can be added at runtime
- We evaluate the usability of our approach based on a theoretical framework and a survey we conducted. Results indicate that our approach does indeed simplify the specification of reactive behavior.
- We show that our implementation parallelizes much better than an equivalent Esper implementation.
- Since users do not want to deal with heterogeneity, we integrate ACTrESS into our middleware, resulting in a middleware that handles event contextualization. We automatically generate the required transformation rules.

1 Related Activities and Publications

The research leading to this thesis was conducted as part of several projects and partial results have been published.

1.4.1 Research Projects

We were involved with the research project DynamoPLV², which is an interdisciplinary research project aiming at integrating production, logistics, traffic and transport. Researchers from operations research, business decision research, production, logistics, traffic and transport management and computer science collaborated in creating various integrated models and approaches on how to integrate data from other disciplines into decisions. The discussions with experts of these disciplines showed the need for real-time processing of sensor data and exposed the challenges we address in this thesis.

1.4.2 Research Collaborations

We were involved in the Dagstuhl seminar³ "Data Warehousing: from Occasional OLAP to Real-time Business Intelligence". We joined the expert group on "Real-Time Data Warehouses and Business Intelligence". In this group, we discussed how to decrease the latency between data entering a system and this data being available in a business intelligence tool. During the discussions we established a deeper understanding of business intelligence requirements, which influenced our overall design.

Within our project in the Software Campus⁴ we collaborated with Software AG⁵. We were able to verify the soundness of our ontology model. Furthermore, we exchanged ideas with Software AG's Research and Development department, to make sure that our ideas have a solid grounding in business reality.

1.4.3 Related Publications

In this section, we want to briefly present our previous work in relation to this thesis. We do not provide details on the work here, as these are addressed in the appropriate parts later in the thesis.

We give an overview of EBSs and how its individual components interact with each other in [37]. The work focuses on integrating event processing with business services. From the discussed material, it motivates the challenge that domain experts should be able to leverage the benefits provided by the event processing paradigm. As mentioned above, this thesis focuses on this aspect. In the context of DynamoPLV, we collaborated with researchers from logistics and business administration to illustrate the benefits and challenges of using the event processing paradigm along the example of supply chains [35]. One of the conclusions we draw is that there is a need for simple, on-the-fly analytics, available to the domain experts.

Our transformation approach, which tackles heterogeneity is presented in [87]. We expand on that idea and detail the mapping between types in [89]. Chapter 4 which introduces ACTrESS is based on these works. A formal treatment of the approach, including a proof for soundness and type safety is presented in [75]. Chapter 5, which describes our event contextualization approach, is based on our previous work presented in [88].

² <http://dynamo-plv.de/>

³ <http://www.dagstuhl.de>

⁴ <http://www.softwarecampus.de>

⁵ <http://www.softwareag.com>

In Section 7.1.2 we show how to use the concept of Eventlets to encapsulate and distribute contextualization tasks. The work on Eventlets is subject of [9, 10, 11]. Eventlets were mainly developed by Stefan Appel in his dissertation [8]. In Section 5.4.5, we use a system called ASIA to distribute monitoring information in a scalable manner. The work on ASIA is discussed in [79, 92, 93, 134]. ASIA is developed in joint work with Sebastian Frischbier, Alessandro Margara, David Eyers, Patrick Eugster and Peter Pietzuch.

We introduce our formalization of sanitizable signatures in [33], which gave the inspiration of suggesting message authenticity as part of the future work.

We also worked on an architecture which promises to significantly shorten the time between data occurrence and availability in data warehouses and business intelligence. We drew the motivation for this from the identified need for on-the-fly analytics. Our approach removes the traditional ETL process and stages raw data directly. Data is then processed on demand based on incoming queries. In addition, we seamlessly weave an event-based component into the architecture, which uses stream processing techniques to provide an overview of the incoming data in a dashboard-like fashion. This allows users to be alerted of certain situations or decide to drill down on the data, for which our on-demand processing is used [90, 207].

1 Structure

This thesis is structured as follows: Chapter 2 presents background information. We provide an overview of publish/subscribe systems and event-based systems. In Chapter 3 we discuss the notion of context in EBSs. We infer a foundation of requirements for using context in EBSs and make a distinction between two kinds of context: context about data and context about events. Chapter 4 proceeds with exploring context about data by providing our transformation framework for EBSs: ACTrESS. In Chapter 5 we introduce a mechanism for context about events. We use policies as high level situation descriptions to progressively contextualize events into more complex events. Both these chapters include an evaluation of our approach. Chapter 6 presents related work, from the perspectives of handling heterogeneity and event contextualization. Finally, Chapter 7 concludes this thesis and discusses future research directions.



2 Background

Technology is the campfire around which we tell our stories.

–Laurie Anderson

In this chapter we provide background information on technologies and systems we used. This also serves the purpose of introducing terminology as we use it throughout the thesis.

2 Publish/Subscribe Systems

The idea of publish/subscribe systems is to provide a scalable communication model to a large number of (distributed) communicating entities. In light of modern IT architectures with tens to thousands of distinct communicating entities, traditional communication techniques like RPC [21] incur severe performance and scalability problems.

Publish/subscribe systems divide communicating entities into publishers and subscribers. *Subscribers* issue subscriptions defining which data they are interested in. *Publishers* publish data in the form of messages. A middleware service (usually called *broker* or *Message-oriented Middleware (MOM)*) matches published messages against issued subscriptions. Figure 2.1 illustrates this. If a message matches a subscription, the service delivers the message to the corresponding subscriber, in form of a *notification*. The distinction between publishers and subscribers is only conceptual, a single software component might have both roles at the same time.

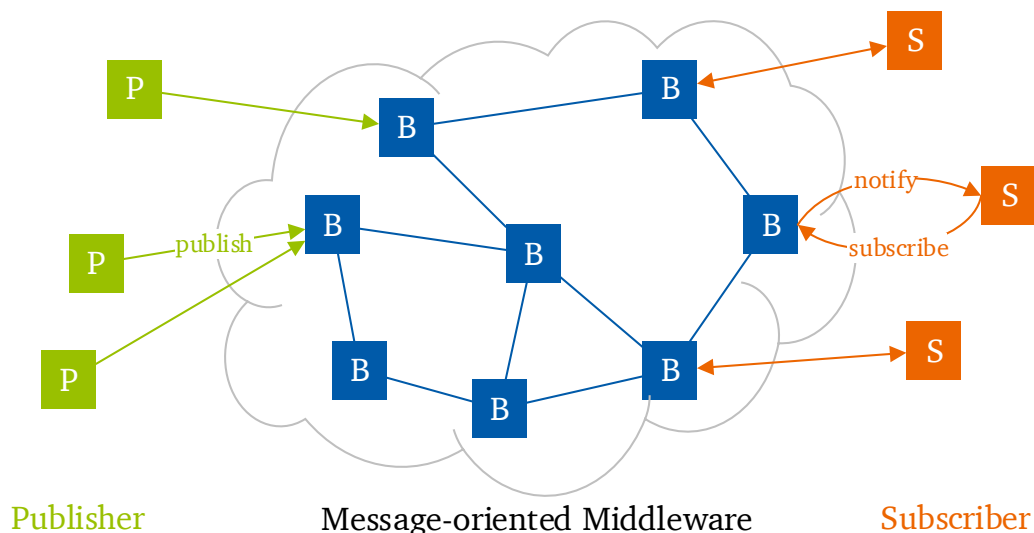


Figure 2.1.: General architecture of publish/subscribe systems. The MOM consists of a distributed network of brokers (B) which forward messages from publishers (P) to subscribers (S)

Some middleware services allow (or even force) publishers to *advertise* their future messages. Advertisements may include the general format of the messages or even promises about their content (e.g., temperatures are always between 0°C and 50°C). This information can be used by the middleware

service to optimize its matching procedure. For example, with the above advertisement and a subscription to messages with temperature $< 0^{\circ}\text{C}$, the middleware service does not have to run the matching algorithm for these messages.

Through this general architecture, publishers and subscribers are decoupled in various dimensions [78]. *Space decoupling* means that publishers and subscribers are anonymous to each other. A subscriber receiving a notification does not know who originally sent the message, and neither do publishers know the recipients of their messages. In fact, publishers and subscribers do not even know how many publishers and subscribers exist. Because of *time decoupling*, publishers and subscribers do not need to be connected at the same time to exchange messages. Publishers may send messages that a currently disconnected subscriber receives sometime later and likewise, subscribers may receive messages sent by a currently offline publisher. *Synchronization decoupling* means that publishing a message does not block publishers and that subscribers are asynchronously notified about incoming notifications. Thus, message exchange is not a synchronous process.

Depending on the expressiveness of subscriptions and the matching mechanism, there are different flavors of publish/subscribe systems:

Channel-based Publish/Subscribe provides the abstraction of channels. Subscribers may subscribe to one or more channels and publishers choose a channel for each message they publish. If a subscriber has subscribed to a channel that a message is published to, the MOM will notify the subscriber about that message. Thus, each channel can be seen as its own (mini-)MOM without any sort of matching. This comparatively simple model was used in early messaging middleware e.g., the CORBA Notification Service [104].

Topic-based Publish/Subscribe is similar to channel-based publish/subscribe in that the main matching criterion is a named channel. In addition, however, subscribers may define filters on the **metadata** of messages. The filter is checked by the middleware and a message is only delivered if all filter criteria are satisfied.

To provide more flexibility, modern topic-based publish/subscribe middleware allow for *hierarchical* topics. A topic T may have a parent T_p (which might have a parent itself). A subscriber subscribing to T_p will also receive messages published to topic T . The Java Message Service (JMS) uses a topic-based approach [66] and while it does not enforce the support of topic hierarchies, the major JMS providers support them.

Type-based Publish/Subscribe works in a similar way to topic-based publish/subscribe with topic hierarchies. Instead of an explicitly specified topic, subscriptions and the matching are based on a message's type. Thus, subscriptions are issued against interfaces a message must realize, rather than strings. A system using type-based publish/subscribe is Hermes [154]. Unlike modern programming languages, type-based publish/subscribe does not assume a single root type and allows for multiple hierarchies to co-exist.

Content-based Publish/Subscribe matches messages based on their content. These systems allow subscribers to express conditions on the message's content in some kind of subscription grammar (e.g., using XPath). Any message that satisfies the specified conditions is delivered to the subscriber. Well-known content-based publish/subscribe systems are Rebeca [143], SIENA [41, 43] and PADRES [80].

Concept-based Publish/Subscribe allows for specifying subscriptions on a semantic level, allowing for different unit systems as supported by Rebeca [52]. In one of its sub-categories, **Ontology-based Publish/Subscribe**, the matching takes synonyms and a concept hierarchy into account,

similar to techniques from the Semantic Web. For example, a subscription to messages containing `university='TU Darmstadt'` would also match a subscription with `school='TU Darmstadt'`. Publish/subscribe systems with an ontology-based approach include OPS [210], Rebeca [7] and S-ToPSS [153].

2 Event-based Systems

Chandy defined an *event* as a significant change in the state of the universe [48]. What is significant is, of course, application-dependent. For some applications, the change in time might already be significant. An event captures a certain state at a certain time. Consequently, events are always discrete instances. Sensors observe signals (e.g., continuous temperature readings, method invocations, or position information) and detect events from these observations. Each observed event is incarnated as an *event object* as Figure 2.2 illustrates. Usually, event objects have an associated *event type*, which defines its structure. For example, an accelerometer sensor turns continuous acceleration data into a discrete event every 10 ms (detection), representing the acceleration at that moment. The corresponding event object has an associated event type, defining the sensor ID, a timestamp and acceleration values for the x,y and z axis as the event type's attributes. The event object corresponding to the detected event carries concrete values for these attributes.

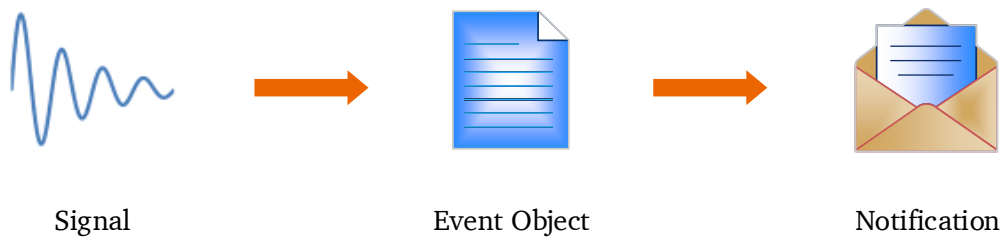


Figure 2.2.: Illustration of event creation

Event producers make event objects available to the rest of the system, by packaging them into a message, thereby turning them into an *event notification*¹. Event-based Systems (EBSs) rely on a *notification service* to transport event notifications from event producers (e.g., sensors) to event consumers (e.g., interested software components) [142].

Typically, this notification service is a publish/subscribe middleware [166]. In fact, many people do not see a clear difference between publish/subscribe systems and EBSs. We believe that the reason for that is that these systems are only seen in terms of their interaction mode, which is push-based where information is pushed to data sinks without a dedicated request for information (sometimes also referred to as *implicit invocations* [160, 195]). However, we see EBSs as more than just an interaction mode. While EBSs always work in a push-based mode, they also include paradigms about processing (Complex Event Processing (CEP)) and reactions (Event-Condition-Action (ECA)), which we will detail below. While many concepts are similar to publish/subscribe systems, we want to make the distinction. EBS rely on a publish/subscribe interaction mechanism to pass meaningful events between components. Events are pushed to interested event consumers who respond to the received event in some way.

Thus, EBSs are sense-and-respond systems [46], usually following the principle of Boyd's famous Observe-Orient-Decide-Act (OODA) cycle [28] or other analogous loops like Monitoring-Analysis-

¹ Please note that we will use 'event' instead of 'event object' and 'event notification' throughout this thesis where it does not lead to ambiguity.

Planning-Execution (MAPE) [206] or Plan-Do-Check-Action (PDCA) [140]. An EBS observes its environment through sensors, which detect meaningful events (*observe*). It then processes the events, deriving high-level events through application of domain semantics (*orient*). After that, the system evaluates the derived knowledge against a set of rules to *decide* about its action. Finally, it carries out the determined action (*act*).

When the event-based paradigm is integrated into larger-scale software, where (complex) events trigger software components, one typically refers to this as an Event-driven Architecture (EDA) [131, 47]. However, EDA and EBS are often used interchangeably (even in the literature). We do attempt to make a clear distinction, but will use EBS when we are referring to the general interaction paradigm and EDA when we are focused on the architectural concerns and implications.

2.2.1 Complex Event Processing

Events produced by sensors often represent very low-level information. For example, the fuel consumption reading of a car's sensor represents only the fuel consumption of the car at a certain point in time. In most applications, these low-level events need to be further processed to extract some meaning from them. For example, to get more stable results, the consumption events should be averaged over the last 2 seconds. The process of combining *simple* events (as produced by sensors) to *complex* events is called *CEP* [130]. There are three basic operations to combine events [110]:

Aggregation Often, events are aggregated according to some aggregation function. Since aggregations only work over a finite set of events and event streams are conceptually infinite, the aggregation always happens over a certain *window* (e.g., the last 5 seconds). Well known aggregation functions are sum, average, maximum, minimum, median and the count of events. This corresponds to the example given above, where velocity events are averaged over the last 2 seconds.

Composition Events can also be combined according to a pattern. For example, the velocity events may be combined with position events to obtain a complex event representing the velocity of the car at a certain position. While aggregation typically only works over homogeneous events, composition can incorporate heterogeneous events.

Derivation The derivation of higher-level events includes some kind of reasoning. Thus, it always requires some sort of domain semantics that are applied. Derivation often also makes use of external knowledge, that is not represented by the participating events. For example, the complex events we obtained by combining velocity and position data may be further correlated with knowledge about speed limits (external knowledge) to warn the driver (domain semantics that cars should not go above speed limits).

Figure 2.3 gives a complex event processing example. It follows the example above where velocity events are first aggregated by applying an average (1). These averaged events are then composed with position events (2) and finally used for the derivation of an even higher-level event (3). As the example shows, complex events can be formed by combining simple or other complex events (or a mix of both).

Event-Condition-Action Rules

The question now is how to tell a CEP system how the desired complex events are formed and what to do upon the detection of certain events. If we see emitting a complex event as just one form of action, we can treat the detection of complex events and reactions uniformly. ECA rules provide a facility to specify these reactions [64]. The event-part specifies a pattern of events necessary to trigger the rule, e.g., a sequence of temperature events $temp_1, temp_2, temp_3$. The conditions-part imposes additional

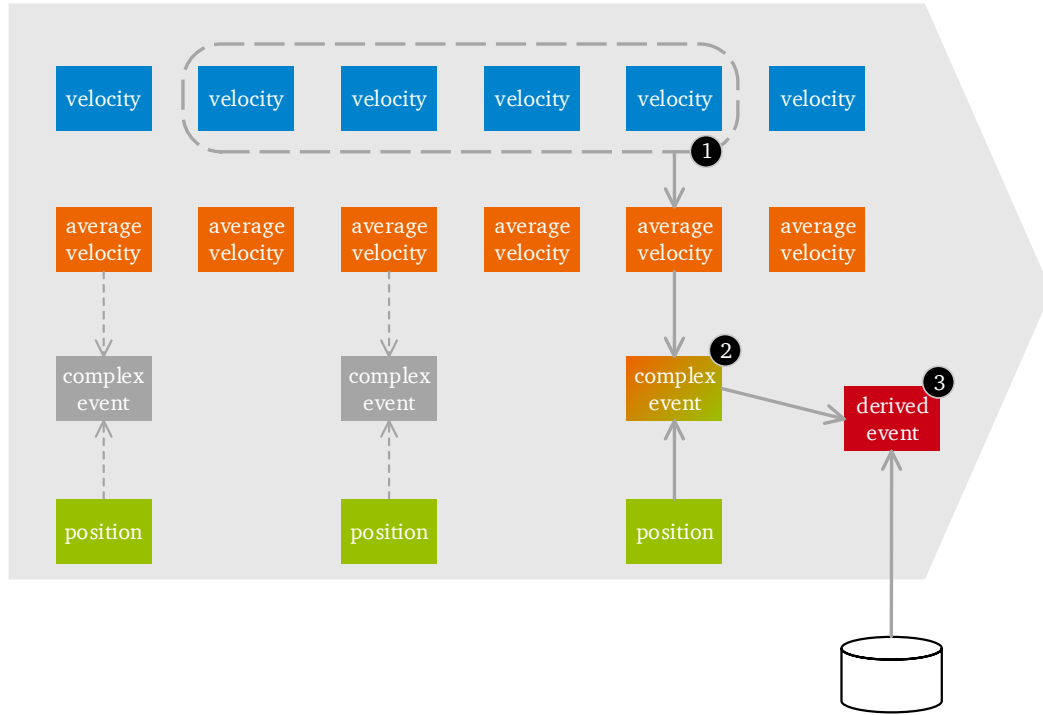


Figure 2.3.: Example of CEP

constraints on valid patterns, e.g., $temp_1.value < temp_2.value < temp_3.value$. Finally, the action-part defines the action to take when a matching pattern is detected (e.g., emitting a complex event or invoking a method). The language to express such patterns and their conditions is called an *event algebra* [45].

Event Algebras define operators for expressing event patterns. The set of operators depends on the concrete event algebra. The HiPAC project for example, defines the following operators [63]:

Disjunction (\vee) A disjunction of two events ($E_1 \vee E_2$) is detected, if either one of the two events occurs.

Sequence ($;$) A sequence of two events ($E_1; E_2$) is detected, if E_2 occurs *after* E_1 has already occurred.

Closure ($*$) The closure of an event (E^*) is detected if the event occurs zero or more times. It is useful to capture multiple occurrences of the same event. For definiteness the closure needs a terminating event: $E_1^*; E_2$.

Other event algebras like that of SAMOS [97] or Snoop [45] extend this operator set, though the additional operators are mostly syntactic sugar and can be expressed by those above. An exception is the NOT-operator. It requires that an event is not detected. Since EBS cannot use the closed world assumption as event patterns define criteria on data that is yet to happen, instead of data that has already been stored, handling the NOT-operator requires some information about the interval during which the NOT-operator is evaluated. For example, $NOT(E; [t_1, t_2])$ is detected, if E does not occur between the timestamps t_1 and t_2 . In this case, explicit timestamps are used. Patterns with events as delimiters are also possible. e.g., $(E_1; NOT(E_2); E_3)$ is detected if an event E_3 occurs after E_1 has occurred, if there was no event E_2 between the occurrences of E_1 and E_3 .

Enrichment

A special form of complex event derivation is adding information to the event. Since the original event is enriched with additional information, this step is called *enrichment*. A common use case is adding

information from a database [110]. For example, when an RFID-badge is scanned at a door, event enrichment adds information about the owner of the badge to the RFID-event. Surprisingly, the Event Processing Technical Society (EPTS) defines enrichment as the process of adding information based on prior events [150] and thus rule out external sources. We think that information from prior events is better handled by aggregation and composition and see event enrichment as a process of adding *external* information to the event.

2.2.2 Cyber-physical Systems

Cyber-physical Systems (CPSs) are one area of application for EBSs. CPSs combine information processing and communication information with monitoring and control of physical processes [197]. The sensed status of physical world processes triggers some computation, whose result influences the process, creating a feedback loop between computer programs and physical processes. An emerging and prominent example are intelligent factories, where parts know where they need to go and route themselves between the factory's machines.

Typically, CPSs incorporate a multitude of sensors and actuators, as well as many individual processing units. This makes EDAs the paradigm of choice when dealing with these kinds of systems. Designing and using CPSs requires understanding the physical processes, computational specifics, communication paradigms and their joint dynamics [68]. This hampers their tremendous potential, as experts in both areas are rare. Thus, as already outlined in Section 1 this thesis suggests an approach to simplify the use of CPSs, by making the software-side easier to use.

3 Context in Event-based Systems

*Getting information off the Internet is like
taking a drink from a fire hydrant.*

–Mitchell Kapor

Capturing, understanding and using context in computer systems is subject to a plethora of research. There are hundreds of books about context-aware systems and even more publications. In this chapter, we explore the role of context in Event-based Systems (EBSs) from a conceptual viewpoint. We will not talk about how context can be derived, as this is subject to individual applications.

We start by reflecting upon existing research about context-aware systems. Using this background knowledge, we derive requirements and general guidelines for context models in EBSs, making a distinction if context describes event data or is itself part of the event data.

3 Context Definition

There is no clear definition of what context is. Depending on the system and its needs, *context* may involve many different things.

Because of this variety, there are different approaches to defining context: Many authors try defining context by enumerating what fits their understanding of context, and what does not [178, 32, 172, 69]. In other definitions, context is often described as the user's/application's environment [31, 211]; other authors define context as the description of a user's/application's situation [86, 161, 112]. These definitions mostly use synonyms (e.g., environment/situation) to describe what context means. Finally, some authors attempt to define context by specifying criteria that a piece of information has to meet to be part of a context [177, 151, 1].

Instead of trying to give another definition of context, we want to focus on an aspect that all the definitions share:

- Context is manifold and depending on the current user or situation, different aspects of the same context are important.
- Context is *assembled* from pieces of information provided by different sensors.
- Context is subject to constant *change*.

3 Context Model

As any kind of system, context-aware systems use a model that determines what these systems can express and reason about. The core part of these models is the context model, which defines structure and semantics of what the context-aware system treats as context.

We make an important distinction regarding the *subject* of context: In EBSs contextual information may be about the event notification or the event itself. Context about event notifications defines the semantics

of the data and helps with the interpretation. A basic example of such context is the unit system which was used for the values in the event. Context about the event itself is information that is not carried by the event, but is still of interest to the application for the current situation. For example, given a position in an event, it is often of interest in which room this position is. Consequently, we call these

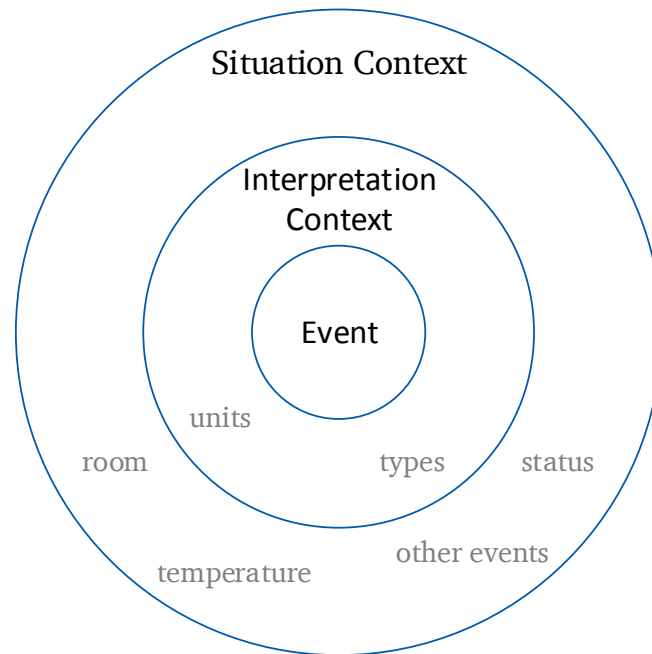


Figure 3.1.: Overview of contexts surrounding an event

two kinds of context that surround an event interpretation context and situation context. Figure 3.1 illustrates this distinction. The interpretation context surrounds the event and is necessary to correctly understand the event. Surrounding both is the situation context, which contains information about the current situation. This also illustrates that without proper interpretation, an event cannot be related to the situation it describes.

We will explore each kind of context in this section.

3.2.1 Context About Data: Interpretation Context

In traditional applications, there is usually an implicit agreement about what data mean. In EBSs however, the decoupling and anonymity between producers and consumers threatens this assumption. Thus, the interpretation of data may differ between components if the assumptions about the data are not shared. We analyzed EBSs and state the following requirements to context models about event data.

Structure

Event-based systems have an internal event representation, called *event model*: There are several ways to represent the structure of an event notification [169]. Notifications can be structurally typed or have no predefined structure. A structure may be flat or hierarchical. If event notifications are typed, their typing system may support inheritance.

A second aspect of structure is the format of event notifications. They can be represented in the Extensible Markup Language (XML), some binary format or any other, suitable format. It is important that

context models can capture both, representation and format, if they want to help with interpreting event data.

Type

Most EBSs use a typed event model. It is thus important to respect types in the context model. Even with identical structural properties, types may differ between participants in an EBS. For example, consider the two events in Figure 3.2. Both events use the same structural properties and are semantically equivalent. However, their types differ in the way the position information is represented. Labels and nesting level differ between the types. Consequently, context models must be able to capture these aspects. Note that sometimes there may not be a one-to-one correspondence between semantically similar types. For example, one type may have an additional attribute, which must be inferred or set to a default value when converting between these types.

```
<PositionEvent>
  <location x="25" y="30" />
</PositionEvent>
```

```
<PositionEvent>
  <position>
    <xValue>25</xValue>
    <yValue>30</yValue>
  </position>
</PositionEvent>
```

Figure 3.2.: Different event types for the same semantics

Value Semantics

Values of events can have very different meanings. For example, in Figure 3.2 it is not clear what 25 for x means. Is it relative to the sensor which detected this information? Is it a Cartesian coordinate or a Polar one? What is the unit of measurement? Usually, this information is not included in the event notifications and must be explicitly captured by context.

Extensibility

EBSs are very dynamic systems where producers and consumers join and leave. Thus, we consider extensibility of the context model as highly important. New clients might use types or a structure never encountered before. In fact, we cannot even claim above context elements to be sufficient for future systems.

3.2.2 Context About Events: Situation Context

We define context about events as any kind of information that is of interest for a given situation and not already contained in the event. For example, given a position event of a person, it might be of interest in which room this person is. Since only the position and not the corresponding room is contained in the event, that room becomes contextual information. For a different situation, however, it may not be the room that is of interest, but the proximity to other people.

Since situations vary across applications we refrain from giving a list of what is context for events as this is highly application-specific. We will, however, develop requirements for context models for EBSs. This is especially useful for event-based middleware, which seeks to support a wide range of concrete event-based applications.

To determine the features the context model should support, we established requirements based on existing surveys and analyzed well-known context-aware systems according to their supported context elements.

Context Elements

As a first step, we looked at existing, well-known context-aware systems to get an overview of their supported context elements. Please note that this is not supposed to serve as related work, since the following analysis helped us in the design of our approach, but the systems have an entirely different goal. By analyzing commonalities and patterns among the systems, we gained an understanding of the things we should support. Table 3.1 shows the results of this analysis.

Project Aura

Project Aura is a runtime environment seeking to integrate various devices (printers, handhelds, computers) with smart environments able to monitor certain activities. It encompasses research on hardware as well as software architecture to allow for creating improved user experiences with less attention required.

To achieve this goal, Aura provides contextual information to end applications, which is mostly focused on currently available devices, network information, information about people and the areas they are in.

CARE

CARE focuses on improving the user experience when using web services, especially with the increasing capabilities of mobile devices. The key idea is that mobile devices lead to a much broader heterogeneity in the contexts of a web service's users, that respecting the user's context when delivering the service can yield great benefits.

CARE's context model thus captures device capabilities, network status, the user's location and speed/velocity.

CoBrA

CoBrA (Context Broker Architecture) targets the ubiquitous computing paradigm. Its premise is that these systems can improve user experience a lot by making use of context. The idea behind CoBrA is to provide a central information broker which provides context to multiple intelligent applications. This enables delivering context information as one coherent model.

CoBrA's context model (COBRA-ONT) focuses on *agents*, which may be software agents or real people. Their location, properties of that location, their role and possible activities are the main parts of the context model.

CoDAMoS

The design of CoDAMoS (Context-Driven Adaptation of Mobile Service) acknowledges that it is hard to build a general and yet suitable context model right away. The authors thus design only basic concepts and allow for extensions to cater to an application's needs.

The five principle concepts are user, environment, platform and information about the service. Each of these concepts encompasses sub-categories and their relations with each other. These can be extended with additional sub-categories or attributes.

COMANTO

COMANTO's idea to separate contextual knowledge into its representation and management. The authors state that prior to COMANTO, generic context models could not cope with frequently changing contextual information. COMANTO brings a generic context model for comparatively static data and a location-based model for context data that must be managed.

Context elements of COMANTO are *profile* information like relationships, physical locations, preferences, etc. In addition, it contains location data and other sensor-read values.

GAIA

GAIA targets context-aware systems from a different angle. Its primary focus is not to delivery contextual information, but to provide an operating system-like infrastructure to help developers make use of context. It thus provides a management infrastructure for various context-enabled devices and services.

GAIA's main focus are active spaces, which are physical locations (e.g., rooms) providing ubiquitous services. As such, its main focus regarding context elements are positional information and information about a specific location.

SOCAM

SCOAM's motivation is a similar one to GAIA. The authors also focus on infrastructure support, providing a middleware which enables developers of context-aware systems to acquire and reason about context. However, the authors also acknowledge that a coherent context model across applications bears many advantages and thus provide the context model that SOCAM exposes to its applications.

SOCAM's context model includes location information, activity information and information about persons.

Analysis Conclusions

While context-aware systems are very diverse and focus on different things, we draw the following conclusion from our analysis:

- Location is supported by all systems and seems to be important. Location-support extends beyond a representation for pure locations and often includes some form of physical relation (e.g., inside, next-to, etc.).
- All systems care for the relationships between entities (e.g., users, services and activities). This *linking* of individual data thus seems to be very important when dealing with context.
- Supporting a large number of applications suggests a middleware approach with a very basic context model.
- Extensibility is important for these general-purpose systems, thus, the model should be extensible.

System	Context Elements
Aura [96]	devices networks people areas attributes: accuracy, confidence, update time, sample interval
CARE [20]	profile and preferences service provider policies user device capabilities user device status network resources user location local time speed content of a service request
CoBrA [49]	location-related (places, buildings, rooms, etc.) agent related (persons, software agents, roles, etc.) agent's location context agent's activity context (events, presentations)
CoDAMoS [156]	user (profile, preferences, mood, activity) environment (time, location, temperature, lighting) platform (processor, memory, bandwidth, available software) service (semantic + syntactic information)
COMANTO [168]	person (person-to-person associations) place (physical location, symbolic/geographic representation) preferences activity agenda (calendar information) time sensors (datatype properties, including their configuration) service (application information) network legal entity (corporate actors)
GAIA [163]	entity (user, application, service, device) physical (location, time) environmental (weather, light and sound levels) informational (stock quotes, sport scores) personal (health, mood, schedule, activity) social (group activity, social relationship, others in the room) application (email, visited websites) system (network traffic, status of printers)
SOCAM [105]	computational entity (service, application, device, network, agent) location (indoor, outdoor) person activity optional domain specific ontology

Table 3.1.: Context elements of various context-aware systems.

Model Structure

In the second step, we focused on how established surveys structure the landscape of context-aware systems. We focused on how these surveys structure the landscape of context-aware systems in order to distill structurally important features. We do not discuss the general conclusions these surveys draw, as they have a different focus. This is important, since there are many, sometimes very different view-points on context-aware systems. We aimed at identifying structurally relevant features of models which contribute to situational awareness.

In "A Survey On Context-aware Systems" Baldauf et al. discuss architecture and context models of context-aware systems in general and give a detailed analysis of existing context-aware frameworks [15]. Among other systems, the main systems they analyze are CASS, CoBrA, the Context Management Framework, the Context Toolkit, CORTEX, Gaia, Hydrogen and SOCAM. They identify six modelling approaches: key-value models, markup scheme models, graphical models, object oriented models, logic based models, and ontology-based models. In addition, they quote researchers who showed that ontology-based models are the most expressive. Generally, context models should be *simple* to use, *flexible and extensible* to support change, *general* so they do not cater to just one use case, and *expressive*.

Bettini et al. discuss requirements for context modeling techniques in "A Survey of Context Modelling and Reasoning Techniques" [19]. In addition, the authors show how to model common context information types using situation descriptions as abstractions. The requirements they state are a) *heterogeneity and mobility* among the context information sources (sensors), b) capturing *relationships and dependencies* between context elements, c) *timeliness* for context history and prognosis support, d) support for *imperfection*, e) support for context *reasoning*, f) the *usability of modeling formalisms*, and g) *efficient context provisioning*. They discuss various modeling techniques similar to those of Baldauf et al., illustrating advantages and disadvantages of each category. For example, ontological models are said to have advantages regarding heterogeneity and representing complex relationships but reasoning causes performance issues. They also describe *situations* as semantic abstractions from low-level context information. For example, a situation can be the co-location of certain people or a specific sound level.

Bolchini et al. focus on context models in "A Data-oriented Survey of Context Models" and introduce an analysis framework for context models [25]. Their analysis framework classifies context models along three dimensions: *modeled aspects*, *representation features*, and *context management and usage*. Modeled aspects include (among others) how space is represented and if the model supports the notion of time. Representation features capture elements about the model itself, for example the type of formalism used and how flexible the model is. Context management and usage includes what kind of reasoning is supported and incompleteness management. Bolchini et al. proceed by categorizing various context-aware applications along these dimensions. The main ones are ACTIVITY, CASS, CoBrA, CoDaMoS, COMANTO, Context-ADDICT, Conceptual-CM, CSCP, EXPDOC, FAWIS, Graphical-CM, HIPS/HyperAudio, MAIS, SCOPES, SOCAM and U-Learn. No two systems use a context model with the same capabilities, illustrating even further that the choice of which kind of model to use is very application specific.

The above analysis strengthened our hypothesis, that we need an ontology-like data model, that allows for representing *relationships* between concepts. However, since performance is critical in event-based architectures, we cannot afford the expressive, yet computationally expensive inference mechanisms like for example OWL-DL provides. We thus require that a context model for EBSs should use a more direct relationship representation mechanism, resulting in a hybrid model.

To bring our established requirements into a structured format, we use the classification provided by Bolchini et al. [25]. Table 3.2 shows the requirements for our context model according to this framework. The left column lists various context elements and the right column shows the result of our analysis regarding context models for EBSs. In the remainder of this section, we want to provide the reasoning for our choices.

	Feature	Analysis Result
Aspects	Space	✓
	Time	✓
	Absolute/relative space and time	both
	Context history	✗
	Subject	application (event)
	User profile	✗
Representation	Type of formalism	key-value-based
	Level of formality	low
	Flexibility	fully general
	Variable context granularity	✓
	Valid context constraints	✗
Management	Context Construction	distributed / at runtime
	Context reasoning	✓
	Context information quality monitoring	hook
	Ambiguity / incompleteness	hook
	Automatic learning	✗
	Multi-Context modeling	✓

Table 3.2.: Requirements for context models EBSs according to the framework by Bolchini et al. [25] as discussed in this section

In terms of context aspects, context models need to support a notion of space and time both in relative and absolute terms, even at the same time. Context histories are not relevant for the model itself. Since we are establishing a context model for events, past information does not need to be part of the model and can instead be captured by referencing past events, for example with event patterns. According to the framework, the subject of the created context can either be the user or the application. Since we are interested in the context of events, we select application as the subject. However, we do not mean the final application built on top of an EBS but rather the event/middleware itself. Since our subject is not the user, we do not need to support user profiles.

The type of formalism in which to ultimately represent contextual data is key-value-based. Events themselves are key-value-pairs with possibly complex values. Thus any kind of data that is added should as well be in a key-value format. Due to the reasons outlined above, mark-up-, graph- or ontology-based approaches are unsuitable. We do not see a reason to impose formal requirements on the model as we are targeting a generic middleware that should work with as many concrete models as possible. Conse-

quently, according to the framework, the level of formality is low. Please note that this does not exclude models with a strong formal foundation, it just states that models are not required to have one. Genericness also forces flexibility to be "fully general"; a focus on just one application domain is infeasible. Likewise, context models must allow for varying granularity of contextual data, since a whole room, or just something/someone in a room may be of equal importance. Context constraints are not needed, since a context is always established around a certain event and is thus rather small in itself.

Contextual data is determined at runtime, to include the most up-to-date information and since EBSs are usually distributed, so must be the context construction. Context models should support some basic reasoning in the context model. For example, given a person and a room, the model must be able to determine if the given person is inside the given room. However, higher-level reasoning-like activity recognition is not needed as this is better performed by Complex Event Processing (CEP). Context information quality monitoring and ambiguity/incompleteness depend on the target application domain. We cannot be certain that they are always needed, nor can we require built-in support for them. Thus, models should support them in a lightweight fashion or provide hooks to support them in the cases where they are needed. Automatic learning is not required, as this usually refers to user preferences, while context for events relates to situations. Context models should support multi-context modeling, as event-based middleware might be run for various modeled domains.

We will introduce our model, which supports all these requirements, in detail in Section 5.1.



Part II.

Concepts and Implementation



4 ACTrESS

*The great myth of our times is that
technology is communication.*

–Libby Larsen

As mentioned in Section 1, a key challenge when using Cyber-physical Systems (CPSs) is heterogeneity. In this chapter, we introduce and evaluate our approach that tackles heterogeneity in Event-based Systems (EBSs) and thus handles context about data (cf. Section 3.2.1).

The problem of data and application integration has been researched intensely for business applications [129] and even commercial solutions like Informatica® exist. However, EBSs have unique properties that prevent easy adaptation of existing solutions and require a new approach to making applications understand each other's data:

- Communication partners are usually anonymous to each other. A consumer cannot rely on a static schema. Thus, interpretation inference based on the producer's identity is cumbersome.
- Producers and consumers join and leave the EBS during runtime and on the fly. Thus, mediation must accommodate a volatile client population. New clients might also introduce new event types, which were unknown before. Given the size of these systems, stopping the entire system to add new components or to just modify existing integration rules is not feasible.
- Given the high rates at which event objects are published, mediation must take place on the fly with low latency.
- Communication in EBSs often happens between partners, without a governing instance. Agreeing on a monolithic, globally accepted data schema, structure and interpretation becomes an infeasible solution in real-world settings [125, 170, 23].
- Mediation between different interpretations must go beyond structural conversion and support the full range of context about data (cf. Section 3.2.1).

Thus, making software components in an EBS understand each other's data requires a new approach. We developed ACTrESS, an EBSs-targeted mediation approach [87]. ACTrESS makes the local interpretation of clients explicit, sends it to the event notification middleware, which then mediates events on the fly. Thus, we add a forth decoupling dimension to publish/subscribe systems: *decoupling in interpretation*.

ACTrESS focuses on event *transformations*. In our approach, we assume that each component resides in its own *interpretation context* that involves a set of parameters (e.g., country, programming language, development team, etc.) which governs how the component interprets event objects. In ACTrESS, an interpretation context comprises (1) *local types* used by the component, (2) *fine-grained declarative transformation rules* specifying the desired transformations between types, (3) *conversion functions* to detail how these transformations are carried out, and (4) *high-level type mappings* making the programmer's intent explicit and being used for verification purposes [89]. Figure 4.1 shows an overview of our approach.

Transformation Challenges

Hinze et al. analyzed application domains and identified a core set of features that are typical for event-based applications [110]. We analyzed those domains with regard to event producers and consumers residing in different interpretation contexts. A mediation mechanism for Event-driven Architectures (EDAs) must handle the following challenges:

Anonymity Matching event notifications to issued subscriptions requires a common interpretation basis and thus mediation, because subscriptions might have been issued with different data interpretations in mind. In EBSs consumers do not necessarily know the identities of producers and vice versa. While it is possible to encode the producer's identity in a notification, depending on a producer's identity violates the publish/subscribe paradigm. Thus, **it is cumbersome or outright impossible for consumers to infer producer interpretation contexts** to interpret the sent data correctly.

Dynamism EBSs support the dynamic joining and leaving of clients. Supporting such dynamicity, however, causes additional challenges for mediation mechanisms. Adapting to joining and leaving clients may require addition, augmentation or changes of interpretation contexts.

Low latency Publish/subscribe middleware systems are often distributed across multiple servers to scale with an increasing number of clients. They aim at avoiding unnecessary communication hops and other overheads to ensure low latency message transmissions. Mediation between different interpretations should not limit scalability nor significantly increase latencies.

Flexibility **It is infeasible to assume a monolithic, global schema with a closed world assumption, to and from which event notifications are transformed.** It is especially impossible to assume such a schema would be known at compilation and be immutable. Mediation approaches must avoid a global schema and support runtime changes to local schemata.

Transformation support Since participants of an EBS are not known a priori, mediation must provide an expressive mechanism for stating a local interpretation and transformation requirements. Without resorting to a programming language, the mechanism should be expressive enough to support mediation between different value semantics, as well as structural changes to event types (cf. Section 3.2.1).

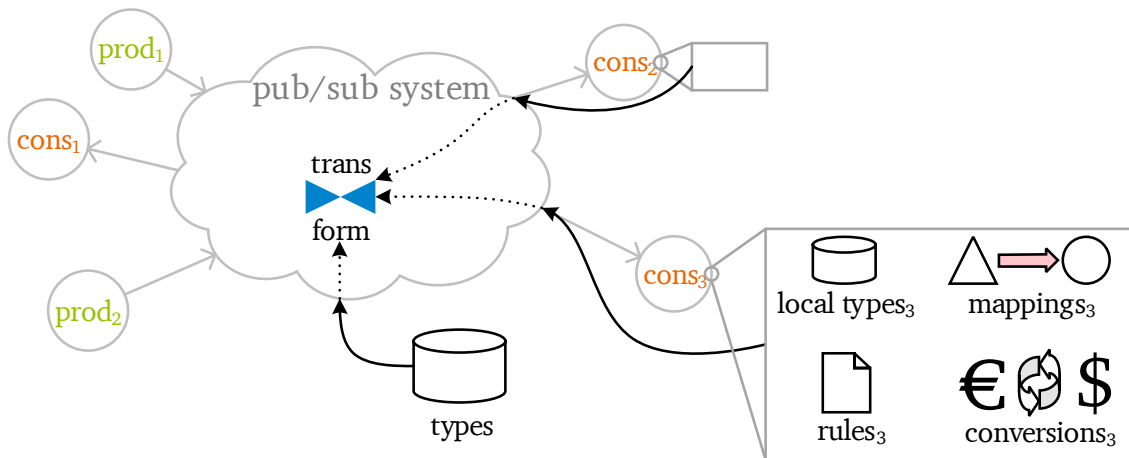


Figure 4.1.: Architectural overview of ACTrESS. Each client supplies its interpretation context (omitted for some clients due to presentation reasons).

In this chapter, we introduce our transformation approach. We start by explaining our transformation framework in detail, including its assumptions and formal definitions. We then show how we implemented this framework and finally provide an evaluation of its usability and performance.

4 Transformation Framework

In this section, we want to detail the fundamentals of our transformation approach, which have been developed in joint work with Patrick Eugster. We introduce supported event representations, the specifics of the transformation rules and the priorities among rules. Furthermore, we identified three dimensions along which we structure the transformation space.

4.1.1 Event Representation

Event-based systems have an internal event representation, called an *event model*: There are several ways to represent the structure of an event notification [169]. Notifications can be structurally typed or have no predefined structure. To allow flexibility even in typed systems, Oki et al. suggest self-describing notification objects [145]. They suggest to employ adapters at the client-side to map between different structures. However, when new producers join the system, several consumers will have to be modified to understand the new structural formats. Some EBS allow defining their event notifications in a hierarchical way, similar to Object-oriented Programming (OOP) [76], while others have a flat, non-hierarchical representation [42]. Notifications can be represented in the Extensible Markup Language (XML), some binary format or any other, suitable format. Each notification has a set of attributes with corresponding values. However, the set of attributes can differ between two applications, even for the same application domain or two different versions of the same application.

To support the vast majority of EBSs, we consider event objects in the general form of typed records of attributes. In accordance with Section 2.2, event objects are instances of a (complex) event type, which we define as follows (\bar{z} denotes a sequence $z_1 \dots z_n$):

Definition 1 (Type). A type T is either a primitive type or a complex type. Complex types are declared as T extends $T_1, \dots, T_n[a_1 : T'_1, \dots, a_n : T'_n]$. \bar{T} are (complex) super-types of T . The attributes of T include those of all its super-types as well as \bar{a} .

This definition allows for arbitrary complex types: in a nested fashion, an event object's attributes may be objects themselves. For example, Figure 4.2 shows `ProductStatusEvent` $[\bar{\delta}]$. The record $[\bar{\delta}]$ is a sequence of the attributes corresponding to `ProductStatusEvent`: `productId`, `pos`, `dynamicPrice`, and `dynamicCost`. Each attribute has a type as defined by `ProductStatusEvent`: `String`, `Position`, `Money`, and `Money`. Since our approach does neither require nor forbid the existence of object methods, we omitted these from the definition.

To support the distributed nature of EBSs, we consider all transferred objects to be values (and not references). Thus, we can apply our approach to other remote communication models with value semantics. Please note that while every event object is always an object, the inverse is not true.

We define the function $attrs(T)$, which returns all attributes \bar{a} of T , along with their respective types \bar{T} . Furthermore, we define the relation $T \preceq T'$, which means that T is a subtype of T' . This might be due to T having been explicitly declared as a subtype of T' , or one of the supertypes of T is a subtype of T' .

¹ Note that typically the Money amount is not encoded as a double. We chose this representation for simplicity.

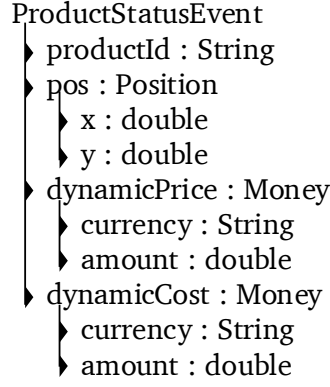


Figure 4.2.: Example event type of a ProductStatusEvent¹

4.1.2 Transformation Rules

We focus on declarative *transformation rules* aiming at minimizing the necessary declarations and aligning well with programmers' mental models. Generally, transformation rules specify which *conversion function* is applied to which attributes in event objects. We process event objects following their nested structure in a top-down fashion.

We want to illustrate this with the example from Figure 4.2: when processing an event object of type ProductStatusEvent we traverse its attributes. During this traversal, we apply any matching transformation rules. Any attributes for which we did not find a matching rule, are traversed unchanged. We follow a depth-first traversal order, e.g., the attributes of position would be traversed before proceeding with dynamicPrice. During traversal, we keep track of the *traversal path*:

Definition 2 (Traversal Path). A traversal path is a 2-tuple $\langle T_0 \cdot \dots \cdot T_n, a_1 \cdot \dots \cdot a_n \rangle$ such that $\forall_i \in [0..n-1], \text{attrs}(T_i) = \langle \dots a_{i+1} \dots, \dots T'_{i+1} \dots \rangle$ and $T_{i+1} \preceq T'_{i+1}$.

For example in Figure 4.2, ProductStatusEvent.Money.amount is a traversal path, but ProductStatusEvent.Position.z is not, because Position does not define an attribute z.

A traversal thus uniquely identifies an attribute within an event type. We deliberately designed the definition so that a prefix of a path is a path as well. For example, if $\langle T_0 \cdot T_1, a_1 \cdot a_2 \rangle$ is a traversal path, then $\langle T_0 \cdot T_1, a_1 \rangle$ is one as well. We do not permit wildcards for traversal paths, but apply them automatically at sensible levels (see *Type Transformations* below).

Separating Patterns and Functions

We need some way of defining which conversion functions to apply to which attributes. As we will see later (cf. Section 4.3.2), this is not necessarily the model in which the user thinks, but is nonetheless required for our formal foundation.

Type Transformations

Since the logistics provider from our example receives product status events from many subcontractors, it cannot rely on a standard format for these events.

Thus, we would like to *default* transformations for certain types (e.g., Position).

Suppose that our logistics provider uses the position information for internal tracking, but also as an added customer service. However, different subcontractors provide the positions in different units and formats (see Section 3.2.1). For example, the origin of the positions may differ or the unit in which the distance to that origin is measured. In addition, the logistics provider wants every position to be in global lat/long coordinates, rather than truck-local x/y positions. Thus, no matter in which event type `Positions` appear, we want to be able to specify a transformation rule to transform all attributes of type `Position` in all event objects. We can express this as:

```
Position ▷ toLatLonCoordinates;
```

Patterns like the above apply at *any nesting level* within a type. E.g., this specific pattern applies to any attribute of type `Position` at any nesting level in any event object. To be more specific about the application, the path on the left hand side may be more specific:

```
ProductStatusEvent.Position ▷ toLatLonCoordinates;
```

This rule applies the `toLatLonCoordinates` conversion only to attributes of `Position` inside `ProductStatusEvents`. Note that the `Position`-attribute may still occur on a deeper nesting level. That is, `ProductStatusEvent.Position` can be seen as `ProductStatusEvent.*Position` where `*` matches any infix, even an empty one. This seems to align well with the original intention of our transformation formation. However, the variability only applies at the last type of a pattern. Thus, something like `TopLevelType.*LevelX.*LevelY` is not allowed.

Attribute Transformations

Specifying transformations based on the type only is sometimes too general. Refer to Figure 4.2 again and consider `ProductStatusEvent`. `ProductStatusEvent` contains two attributes of type `Money`. Assume that `dynamicPrice` represents a time-dependent price for the delivery and should be converted to the customer's currency. On the other hand, `dynamicCost` represents the current cost of delivery, which may depend on environmental conditions, e.g., increased cooling costs in hot weather. We want to be able to treat the `Money`-attributes differently.

Among references to types, we also support references to attributes in transformation rules:

```
ProductStatusEvent.dynamicPrice ▷ toDollars;
```

This rule applies the `toDollars` conversion function only to the `dynamicPrice` attribute in event objects of type `ProductStatusEvent`. Any other attributes of type `Money` are not affected by this rule.

Subtyping Support

Most modern OOP languages support a notion of subtyping. We focus on nominal subtyping [144], as the major programming languages support that. Thus, any event object may carry (at any nesting level) an attribute a with dynamic type T' , even though the corresponding event type defines the type T for a , if $T \preceq T'$. Thus, we want to enable our transformation rules to be subtype-sensitive. Suppose we define a type `PositionWithHeight` extending `Position` (`PositionWithHeight \preceq Position`). `PositionWithHeight` contains an additional attribute z . We can then define a rule

```
ProductStatusEvent.position(PositionWithHeight) ▷ ...;
```

which applies only for `position` attributes whose dynamic type is `PositionWithHeight` or any of its subtypes.

Patterns and Rules

We can now extend our definition of a traversal path (Definition 2) to that of a pattern:

Definition 3 (Pattern). A pattern $p = T.q_1 \cdots .q_n$ consists of a type reference T followed by a (possibly empty) sequence of qualifiers, and denotes attributes in event objects. A qualifier refers either to all attributes with a given type (type qualifier T) or to a given attribute a with a given type T (attribute qualifier $a(T)$).

We only permit *valid* patterns and require that the following conditions hold for all prefixes $T.q_1 \cdots .q_i.q_{i+1}$ (with T_i denoting the type of q_i):

- if q_{i+1} is a type qualifier T_{i+1} : then T_i contains at least one attribute of type T_{i+1}
- if q_{i+1} is an attribute qualifier $a_{i+1}(T_{i+1})$: then T_i contains an attribute a_{i+1} of type T'_{i+1} and $T_{i+1} \preceq T'_{i+1}$.

Attribute qualifiers may be declared without explicitly stating their type. In this case, we assume the type according to event the type definition containing the attribute.

Finally, we can define transformation rules:

Definition 4 (Transformation rule). A transformation rule is of the form $p \triangleright f$ where p is a pattern identifying a set of attributes and f refers to a conversion function.

Conversion functions are used to transform attributes at any traversal path to which a pattern applies. Since conversion functions are defined separately from transformation rules, they can be reused in multiple rules. We believe that this improves the ease of use and reusability of both rules and functions. The conversion function takes as an argument an object of the attribute's type as identified by the pattern and returns an object that fits the new type according to the mapping (see Section 4.2). For example,

`Position \triangleright toLatLonCoordinates;`

refers to a function of the form

```
LatLonPosition toLatLonCoordinates(Position pos) {  
    // convert here  
    return new LatLonPosition (...);  
}
```

Type changes (e.g., from `Position` to `LatLonPosition`) are defined and resolved through mappings, which are explained in Section 4.2).

The example also illustrates that conversion functions may be lossy. However, a client that wants to have `Positions` converted to `LatLonPositions` can most likely deal with the decrease in precision. If they need a higher precision, they will not have `LatLonPosition` in their data model. The inverse –transforming from `LatLonPosition` to the more precise `Position`– may have to employ techniques to improve precision (e.g., combine multiple positions). Our model enables this by supporting stateful transformation.

Rule Priorities and Conflict Resolution

Following definition 3 and 4 it is easy to produce conflicting rules. Rules conflict if their patterns can match the same part of an event. For example, the rules

$r_1 = \text{Position} \triangleright \text{toLatLonCoordinates};$
 $r_2 = \text{DangerCheck.Position} \triangleright \text{toIdentity};$

conflict, because for attributes of type `Position` in events of type `TruckLocal` both patterns match. A simple resolution strategy would be based on declaration order of rules. However, this can quickly lead to unintentional mistakes when rule sets are anything but small. In fact, being able to specify exceptions for special cases is a desirable property. As in the example above, the intention is to generally transform all `Positions` to latitude/longitude coordinates, except those meant to check for possible dangers. We thus introduce rule priorities, following the intuition that more precise rules should take precedence over less precise ones. If our priorities lead to rules with equal priorities, we default to the last rule specified and issue a warning to the user (cf. Section 4.3).

Nesting Level

In some cases, we want to transform certain attributes in *specific* (event) types with a different conversion function. For example, our logistics provider checks certain security constraints like a minimum distance between two potentially dangerous substances. These checks do not need a conversion to latitude/longitude. Thus, `Positions` in `DangerCheck` events should not be transformed.

As in the example above, we would like rules with *more specific* patterns to override rules with less specific patterns. More specific in this case refers to the length of pattern, or, *nesting level*.

If both above rules are combined in one rule set, r_2 overrides r_1 for `DangerCheck` events. Any other attributes of type `Position` are transformed according to r_1 (using `toLatLonCoordinates`).

Instances Over Types

As indicated above, `ProductStatusEvent.dynamicCost` should not be converted to dollars, unlike other attributes of type `Money`. A rule set containing

$r_1 = \text{ProductStatusEvent.dynamicCost} \triangleright \text{toIdentity};$
 $r_2 = \text{Money} \triangleright \text{toDollars};$

achieves exactly that. Any attribute of type `Money` is converted with `toDollars`, except for the `dynamicCost` attribute in `ProductStatusEvents`. The reasoning of this precedence is the prioritization of *instances over types*. Thus, we prioritize attribute-level declarations over type-level declarations. Please note that rule r_1 would still take precedence, even if rule r_2 's pattern was `ProductStatusEvent.Money`.

Subtyping Level

Since the declaration of a specific subtype in a rule's pattern makes the rule more specific, we follow the natural intuition of using the *subtyping level* as an indicator for precedence. Among conflicting rules, we pick the one which refers to the most *derived* type. For example, among the rules

$r_1 = \text{ProductStatusEvent.position}(\text{PositionWithHeight}) \triangleright \dots;$
 $r_2 = \text{ProductStatusEvent.position} \triangleright \dots;$

we pick r_1 since $\text{PositionWithHeight} \preceq \text{Position}$ (the pattern $\text{ProductStatusEvent.position}$ is the short version of $\text{ProductStatusEvent.position}(\text{Position})$). Please note that, unlike dynamic method dispatching [5], rule selection is still static and thus does not impact runtime performance.

Precedence Rules

The three ideas on precedence given above may conflict among each other. For example, the pattern Type1.Type2.Type3 has a deeper nesting level than Type1.attribute , but no attribute qualifier. To have a solid foundation, we list the full set of precedence rules, which allow for unambiguous transformation rule resolution. Rules P1-P3 summarize and formalize above intuitions and rules P4-P6 resolve conflicts between these.

- P1 Nesting level: A deeper nesting level represents a more specific pattern and should thus take precedence over a pattern with a less deep nesting level: a transformation rule with the pattern $T_0.T_1$ will be chosen over one with the pattern T_1 for attributes of type T_1 at a traversal path rooted at type T_0 .
- P2 Instances over types: We give attribute qualifiers in a pattern precedence over type qualifiers. Thus, a pattern $T_0.a_1(T_1)$ is chosen over $T_0.T_1$ for an attribute a_1 in an event of type T_0 .
- P3 Subtyping level: Our third intuitive choice of precedence prefers patterns which use the most derived types: if T'_1 is a strict subtype of T_1 , then a pattern $T_0.a_1(T'_1)$ is chosen over $T_0.a_1(T_1)$.
- P4 Subtyping order: Many OOP languages, including major ones like Java, C++ and C#, support types with *multiple* super-types. This may lead to ties in rule resolution, similar to those encountered in other cases of multiple inheritance (e.g., method selection). Consider for example the patterns $p_1 = T_0.T'_1$ and $p_2 = T_0.T''_1$ which both apply to a traversal path $\langle T_0.T_1, a_1 \rangle$, where T_1 is a subtype of both T'_1 and T''_1 . If the subtyping level of T_1 differs with respect to T'_1 and T''_1 , P3 takes over. If they are the same, we use the order of the subtyping declarations for breaking the tie, i.e., if T_1 extends T'_1, T''_1, \dots , rule p_1 is chosen.
- P5 Instances over nesting level: Since P1 and P2 may conflict, we break ties between them: Consider the patterns $p_1 = T_0.a_1(T_1).T_3$ and $p_2 = T_0.T_1.T_2.T_3$. Both patterns match the traversal path $\langle T_0 \dots T_3, a_1.a_2.a_3 \rangle$. According to P2, p_1 is more specific, but according to P3 p_2 should be preferred due to its deeper nesting level. However, the prefix $T_0.a_1(T_1)$ is more specific than the prefix $T_0.T_1$ and thus, p_1 is prioritized.
- P6 Subtypes over instances: We break ties between P1 and P3 as well as between P2 and P3 by letting subtyping take precedence in both cases. We first check for subtypes on a qualifier's type and only thereafter evaluate if the qualifier refers to an attribute or a type. Thus, among the patterns $p_1 = T_0.T_1$ and $p_2 = T_0.a_1(T'_1)$ we prioritize p_1 if and only if T_1 is a strict subtype of T'_1 .

Our technical report elaborates on transformation rule resolution, including a formal characterization of the resolution semantics and arguments for its type safety [75].

4.1.3 Transformation Space

Event transformations can be of different complexity. A very simple form of event object transformation is the modification of primitive attribute values. These values often have a unit (implicitly) associated with them, e.g., money types (cf. $\text{ProductStatusEvent}$ in Figure 4.2). In this case, the transformation

results in unit conversion. A similar form of transformation is the conversion of primitive values between different architectures or platforms. A far more complex form of transformation is the modification of an object's internal structure. This may involve merging attributes, eliminating attributes or even create new ones based on some information (cf. Figure 4.7).

To get a better understanding of the transformation space we are dealing with, we structured this space along three dimensions: *granularity*, *expressiveness* and *topology*. Analyzing these dimensions helped us support the most complex transformations, while enabling performance optimizations for less complex transformations. The categorization is further useful for structuring related work.

Granularity indicates the minimal relative size of what is transformed. We identified four categories (see Figure 4.3 for an illustration):

- G1: Monolithic object transformations: Event notifications as a whole are subject to transformations.
- G2: Attribute-wise transformations: The source event type and the transformation result have the same number of attributes and attributes are mapped in a 1-to-1 fashion.
- G3: Nested attribute-wise transformations: As G2, this category maps attributes 1-to-1, but supports attributes that are non-primitive types. Thus, nested structures are supported.
- G4: Path-based transformations: Transformations can be expressed on arbitrary attributes at any nesting level of the event type.

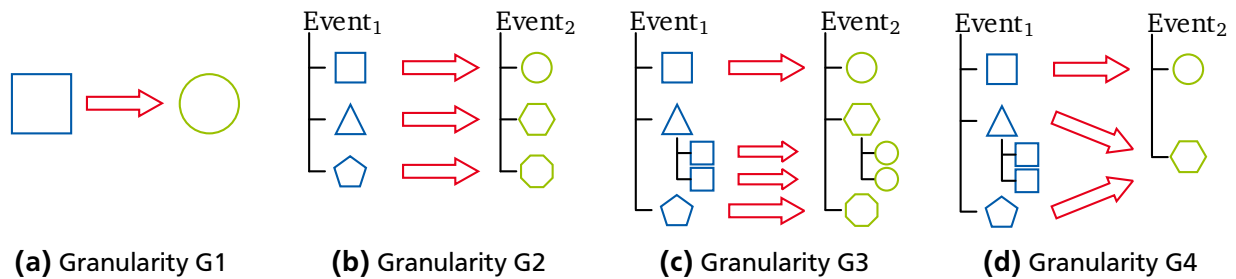


Figure 4.3.: Illustration of the transformation space granularity

Transformations can exhibit various levels of **expressiveness**, which we categorize as follows:

- E1: Type (or metadata) transformations: The content of an event notification is left unchanged. The transformation changes the event type, for example subtype subsumption, where only the super-type's subset of the original attributes are retained.
- E2: Lookup-based transformations: Values (primitive or complex attributes) are substituted by the result of a lookup process. This process may use static or dynamic data structures to obtain its results. However, the lookup always responds with discrete values.
- E3: Function-based transformations: The new value for an attribute is the result of invoking a function, which may perform any computations to construct the new value.
- E4: Function-based stateful transformations: In addition to E3, transformation functions may persist state in variables.

The actual place in the **topology** of the event-driven architecture where the transformation happens creates another dimension:

T1: Peer-based transformations: Each application component (e.g., event producers or consumers) is responsible for transforming incoming/outgoing event objects.

T2: Coordinated transformations: A potentially distributed middleware invokes a dedicated server or software component to transform event objects it passes along.

T3: Decentralized transformations: In this category, the distributed middleware transforms event objects without depending on a centralized component.

Our approach supports the highest level in all of these dimensions: we support path-based (G4), function-based stateful (E4), decentralized (T3) transformations.

4 Interpretation Contexts

As mentioned in Section 4, producers and consumers have an interpretation context, which governs their understanding of events. In ACTrESS, we make these interpretation contexts explicit. Thus, upon connecting, producers and consumers can send their respective interpretation context to the broker, which then handles transformations transparently.

We propose specifying interpretation contexts in relation to another interpretation context, their *parent*. A root interpretation context serves as an anchor point, similar to [52]. Such an approach greatly reduces the $n \times m$ complexity of direct mappings between each interpretation context would incur. Since our approach does not require a globally accepted root interpretation context, it is possible that each broker uses its own definition. Thus, in theory, we do not eliminate $n \times m$ complexity. Since the number of producers and consumers usually exceeds the number of brokers by orders of magnitude, we still believe that this approach greatly reduces complexity. We do not require the root interpretation context to be setup before operation starts. In fact, it can be extended during runtime and we even support an empty root to start with.

We do not require each producer or consumer to define their own interpretation context. They may instead reference an existing context by its unique identifier. This is useful for example when a single software module makes use of multiple producers and consumers. Most likely, all of them will have the same understanding of events and can thus share an interpretation context.

Since an interpretation context's parent does not need to be the root, we allow for arbitrarily complex hierarchies, similar to type hierarchies in OOP languages (see Figure 4.4 for an example). This keeps specification and maintenance efforts to a minimum. Even if a client's interpretation context's parent is not the root context, this client does not depend on the presence or identity of another client. Interpretation contexts are retained separate to a client's connectivity.

We build on our transformation framework from Section 4.1 and define interpretation contexts (see Figure 4.5 for an example):

Definition 5 (Interpretation Context). An interpretation context is a 4-tuple $(\bar{t}, \bar{m}, \bar{r}, \bar{f})$ where

- \bar{t} is a set of type definitions (cf. Definition 1)
- \bar{m} is a set of type mappings of the form $T \Rightarrow^h T'$, where $h \in \{?, !\}$ indicates transformations of incoming/outgoing objects respectively
- \bar{r} is a set of transformation rules (cf. Definition 4)
- \bar{f} is a set of conversion functions $f : T \rightarrow T'$, transforming instances of T to T'

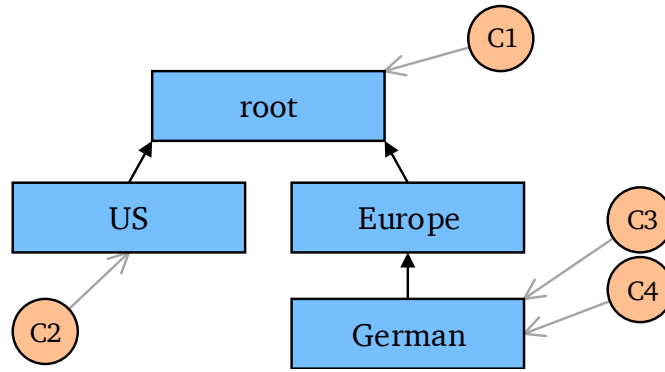


Figure 4.4.: Interpretation context example, with contexts called root, US, Europe and German. Client C1 has been defaulted to the root. C2 has US, while C3 and C4 both have German, which inherits all definitions of the Europe.

Types	Transformation Rules	Conversion Functions
<div>ProductStatusEvent</div> <ul style="list-style-type: none"> ▶ productId : String ▶ pos : Position <ul style="list-style-type: none"> ▶ x : double ▶ y : double ▶ dynamicPrice : Money <ul style="list-style-type: none"> ▶ currency : String ▶ amount : double ▶ dynamicCost : Money <ul style="list-style-type: none"> ▶ currency : String ▶ amount : double 	<div>Mappings</div>	<div> meters-to-yards = value * 1.09 meters-to-feet = value * 3.28 meters-to-inch = value * 39.37 toidentity = value : </div>

Figure 4.5.: Example root interpretation context

In heterogeneous systems, it is infeasible to assume a common type system (e.g., Java). And even within one type system different software components might use different types to represent the same information albeit with different interpretations. **To ensure compatibility with the client's type system, clients can include their specific event types as part of the interpretation context.**

We discussed transformation rule sets in the previous section. Conversion functions define how the actual conversion of attributes or entire events is carried out. We did not specify their signatures in the previous section. Generally, these functions may return values of a different type than that of their formal argument. This allows for type conversions, which is desired when clients use differing types / type systems. Conversion functions may thus change the event type of transformed events. For example, `toLatLonCoordinates` converts from `Position` to `LatLonPosition` and consequently the event type `ProductStatusEvent` is changed to `CustomerProductStatusEvent`. Generally speaking, if the type of any attribute (at any nesting level) is changed, then the entire event type changes.

To capture these type changes, we include type mappings in interpretation contexts. A type mapping $m = T \Rightarrow^h T'$ indicates that the event type T is mapped to the event type T' , i.e., applying the transformation on events of type T results in events of type T' . This allows for using mappings for type checking. h specifies the *direction* of the mapping. That is, if the mapping applies to received ($h = ?$) or sent ($h = !$) events. Please note that this specification is from the broker's point of view.

There is not necessarily a 1:1 relationship between mappings and rules. For example, multiple rules might affect different attributes of the same event type and thus be captured by the same mapping.

Inversely, a single rule like `Position ▷ toLatLonCoordinates` may affect multiple event types and thus require several mappings.

For example, the mapping `ProductStatusEvent \Rightarrow^h CustomerProductStatusEvent` in Figure 4.6 specifies the mapping of `ProductStatusEvents` to their US interpretation context counterpart. By decoupling the specification of *which* types are mapped to which other types (\overline{m}) from *how* this happens (\overline{r}), we further improve flexibility.

4.2.1 Interpretation Context Specialization

We have said that interpretation contexts are arranged in a hierarchy similar to the notion of inheritance in OOP languages. However, we have not yet mentioned what the exact semantics of this *specialization* is. We thus elaborate on the meaning of this specialization in the following, with respect to inheritance and the meaning for transformation.

Inheritance

To make defining new interpretation contexts as easy as possible, a child context only needs to specify the differences to its parent. This allows clients to reuse existing definitions easily. More precisely, inheritance of event types, mappings, rules and functions between a parent e_0 and a child interpretation context e_1 (e.g., between `Europe` and `German` in Figure 4.4) obeys the following rules:

- *Event types* are straightforwardly inherited by e_1 . Note that having more types available than needed by a client of e_1 is no problem, as additional types can be simply ignored by that client.
- *Type mappings* are inherited from e_0 but e_1 may override them: if e_0 defines a mapping $T \Rightarrow^h T'$, then e_1 can override (redefine) the mapping to $T \Rightarrow^h T'$. Mappings can be *overloaded* through subtype-sensitivity of mappings. For example, for two types T and T' , with T' subtype of T , a mapping $T' \Rightarrow^h \dots$ takes precedence over $T \Rightarrow^h \dots$ for instances of T' .
- *Transformation rules* are inherited with the option of being overridden as well. Overriding only happens for rules with *identical* patterns. Rules with different patterns co-exist and are subject to the priorities listed in Section 4.1.2.
- *Conversion functions* are inherited and may as well be overridden, although this is unlikely. If e_1 wants to override a conversion function of e_0 , it must specify a function with the same signature.

The root interpretation context (see Figure 4.4) should represent all reference types, which are thereby available to other interpretation contexts. In a multi-language setup, these reference types are most likely specified in an independent declaration language.

We do not enforce the presence of mappings and transformations for all types to/from the root interpretation context. Sometimes, there may be no semantically sensible transformation, or types are not needed by that particular software component. However, the absence of a mapping is detectable when analyzing an interpretation context's specification and we signal this to the client (i.e., through an observation message). We will elaborate on warning and error messages when discussing our implementation in Section 4.3.

Figure 4.6 shows an example of an interpretation context specialization. The US context extends the root and specifies `CustomerProductStatusEvent` among other custom types, and a mapping to `ProductStatusEvent` from the root. Furthermore $\{r_1, r_2, r_3\}$ is the set of transformation rules. r_1 and r_2 refer to conversion functions included in the US interpretation context, while r_3 references `toIdentity` from

the root, which is accessible as well. These definitions lead to the sample transformation depicted in Figure 4.7. Note that the transformation of positions transforms from relative positions to absolute positions. The reference point for this transformation can be kept as state in the conversion function. If the reference point changes, for example because it is a moving truck, the function itself may be an event subscriber, to be informed about the truck's current position.

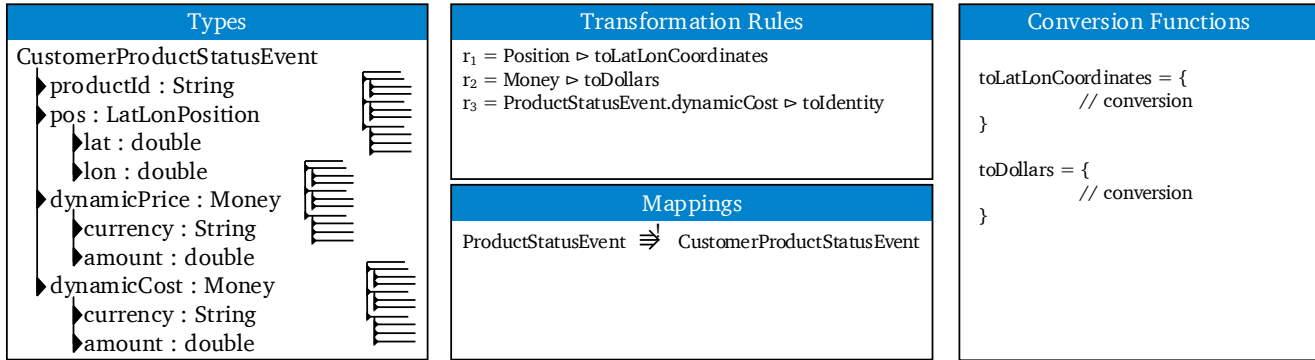


Figure 4.6.: US interpretation context

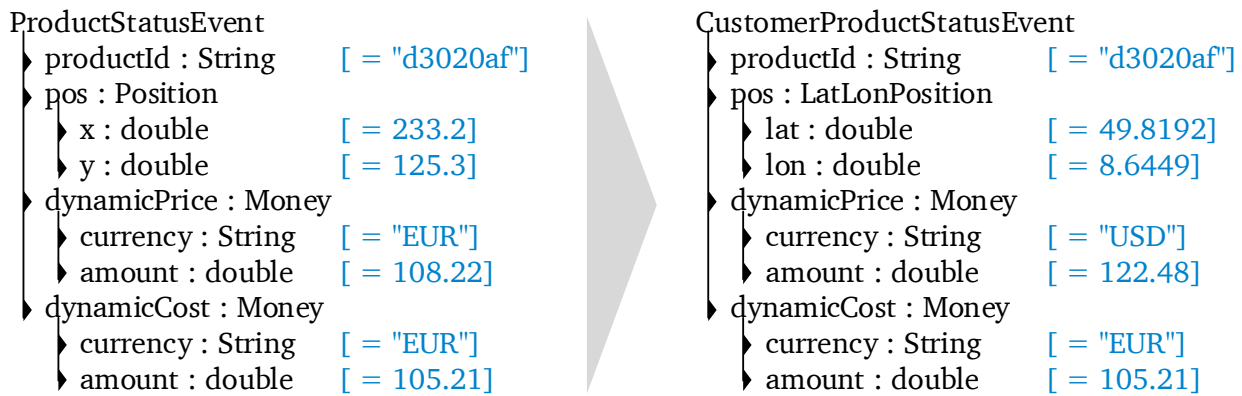


Figure 4.7.: Sample transformation. ProductStatusEvent and CustomerProductStatusEvent are context-specific interpretations of the same object. Note that currency is typically encoded explicitly, but units for other values are almost never encoded.

Transformation

Despite a possibly deep interpretation context hierarchy, resulting in a chain of context specializations, the transformations themselves occur always *directly* to and from the root, as shown in Figure 4.8. Thus, events produced in a certain interpretation context (e.g., German) are directly transformed to the root. To avoid unnecessary work, we only transform if the event is needed in another interpretation context. Events in the root interpretation context are transformed directly to the target context, except to the original one, e.g., German.

Indirect transformations that follow the specialization chain are also possible, but lead to unnecessary delays. Our model is designed in such a way that interpretation contexts can be analyzed upon creation and allows for the generation of direct transformation components. This happens by basically analyzing which types, mappings, rules and functions are visible in the current interpretation context and then creating the transformation components as if this context's parent was the root.

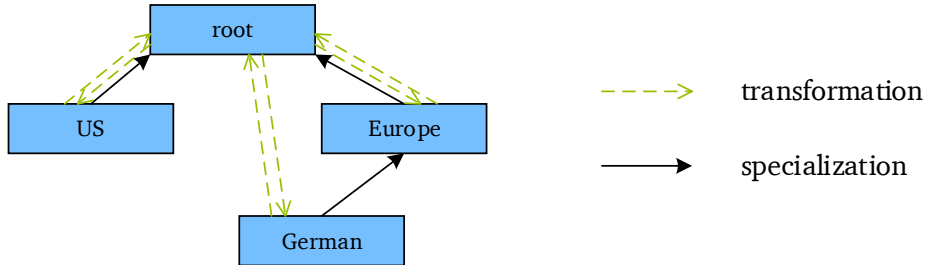


Figure 4.8.: Interpretation context specialization and resulting transformation paths.

4.2.2 Augmenting Interpretation Contexts

In Section 4 we argue that transformation approaches must be adaptable at runtime. To support that, our design allows for easy and seamless augmentation of the root interpretation context with new types, rules and conversion functions. Technically, we also support the addition of mappings, but we have not yet encountered a case where the mappings in the root are non-empty. New event types can be added to the root interpretation context at runtime. Existing interpretation contexts will not be affected, since their transformation rules do not refer to the new types. New conversion functions can be added as well. Even if a new function is named exactly like a function of an extending interpretation context e_1 , this will not change how e_1 works, as the new function is simply overridden.

Adding transformation rules is not straightforward. Consider the case where the interpretation context to be augmented has a child context with a rule $r_1 = T_1 \triangleright f_1$. If we want to augment the parent context with a new rule $r_2 = T.T_1 \triangleright f_1 0$, the child will inherit r_2 , which takes precedence over r_1 for attributes with a traversal path of $T.T_1$. Thus, in this case, we inject a rule $r'_2 = T.T_1 \triangleright f_1$ into the child context, which overrides r_2 and makes sure the original behavior is maintained.

Augmenting usually happens to accommodate new producers and consumers joining the broker network, which use new event types. Since our approach transforms events in the middleware, other clients are not affected.

4.2.3 Modifying Existing Interpretation Contexts

Most changes can be handled by defining new interpretation contexts or augmenting existing ones. However, sometimes it may become necessary to modify existing parts of an interpretation context, for example due to initially unknown requirements. In the extreme case this means changing existing parts of the root context. For example, new address information might be needed by several producers and consumers, which is not yet captured by the **Address** type in the root interpretation context. Only event types are subject to modifications, since they represent some kind of agreement between interpretation contexts. Mappings, rules and functions are context-specific and can thus simply be overloaded, if need be.

If there is currently no client using the interpretation context which is to be modified, that is, no client uses the context or any of its descendants, the modification can happen trivially. In all other cases, we enable this modification in the following way: if a client wants to substitute an existing type T by a new type T' it may do so as long as (1) T' provides all the attributes that T does, and (b) it supplies appropriate transformation functions $f_1 : T \rightarrow T'$ and $f_2 : T' \rightarrow T$. f_1 may do some inference or just set the additional attributes to null. During transformation composition, f_1 is automatically chained to any transformation $g : * \rightarrow T$ resulting in $g' = g \circ f_1$. Similarly, f_2 is chained with any function $h : T \rightarrow *$ resulting in $h' : f_2 \circ h$. Existing interpretation contexts which are agnostic to the change are

thus unaffected. Clients and contexts may be updated over time to accommodate the new type, but this can be done asynchronously.

4 ACTrESS Architecture and Implementation

This section presents our implementation of the transformation model introduced above. We give details on rule resolution, error handling upon the detection of invalid rules and showcase how our approach allows for generation of static code to improve performance. We implemented our approach in Java as an ActiveMQ [189] plugin. Our implementation addresses all challenges outlined in Section 4.

4.3.1 Architecture

We suggest the following architecture for distributed message brokers like Siena [42], Padres [80], Hermes [154] or peer-to-peer based approaches [199], as well as for more centralized approaches like Java Message Service (JMS)-brokers such as ActiveMQ [189]: brokers are enhanced with an *Interpretation Handler*, resulting in the architecture shown in Figure 4.9.

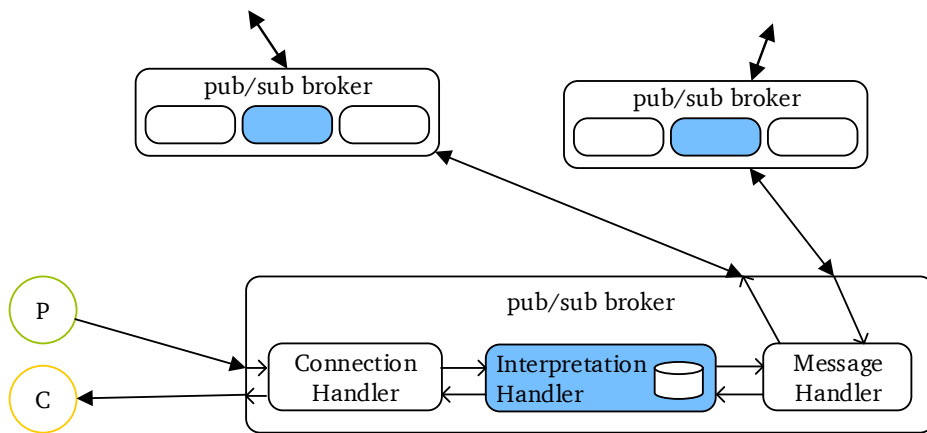


Figure 4.9.: Abstract architecture of our implementation.

Publish/subscribe brokers typically have a *Connection Handler* component which handles connections to producers and consumers, forwarding incoming messages to a *Message Handler*. The Message Handler then does the routing. Similarly, messages intended for subscribers arrive at the Message Handler, which passes the message to the Interpretation Handler, before it is actually sent to the consumer. This enables the Interpretation Handler to intercept and transform messages. The Interpretation Handler stores which client is using which interpretation context to avoid repeated sending of context definitions.

Upon receiving a message from a producer (P), the Interpretation Handler transforms the message into the root interpretation context, according to P's interpretation context. After the transformation, the message is passed to the Message Handler and the middleware can process the message further, e.g., compute the matching against subscriptions. Once a message reaches the fringe of the broker network and is about to be sent to the consumer (C), the Interpretation Handler of that broker node transforms the message from the root interpretation context into C's interpretation context. Similarly, subscriptions are transformed to the root interpretation context, so that each consumer may issue subscriptions in its own interpretation context. Thus, we do not have to modify the matching algorithm of the broker, and can yet correctly handle messages and subscriptions from different interpretations.

Only participants of the same broker network need to establish a root interpretation context — we do not require a global a priori agreement. Different root contexts can be used in different broker networks. Messages can even be passed between these broker networks by the same means as messages are exchanged between a broker network and its clients. Note that enforcing a globally accepted (albeit flexible) root interpretation context would reduce complexity even further, but does not reflect typical distributed, event-based applications in which there is no central controlling unit.

While we advocate transforming in the middleware (i.e., on broker nodes; see next Section), our approach does not require the presence of brokers. Optimizations which bypass dedicated brokers between co-located clients [117] are applicable to our approach for example by putting the transformation code in client-side libraries.

Interpretation Context Repository

The Interpretation Handler maintains an *Interpretation Context Repository*, which stores the definitions of all interpretation contexts known to the broker. Contexts are identified by a unique identifier. Since each broker node keeps its own repository, it is possible for each repository to contain differing contents. This is useful, as only the necessary interpretation contexts need to be stored. However, it might be desirable for consistency reasons or dynamic load balancing to have a synchronized repository across all nodes. Any suitable algorithm for data exchange in a distributed storage can be used for that [124, 180]. As we have shown in previous work, we can integrate the necessary maintenance messages into the regular network traffic with little overhead [79, 93].

We believe that keeping the interpretation context repository synchronized across all broker nodes is the preferable option. Compared to the rate of events, the rate at which producers and consumers join and leave the system is low. In addition, producers and consumers usually only define a new interpretation context when joining the system *for the first time*, reducing the rate at which synchronizations have to occur to a minimum. The benefit of always synchronizing is that clients can connect to a different broker node (e.g., due to locality or load balancing) the next time they join the system.

If the middleware uses channel-based routing, transformations can be performed on any node along the routing path, allowing for highly flexible load balancing. However, nodes that want to transform a message must be able to obtain knowledge about the producer’s or the consumer’s interpretation context. If the repository contains the required definition, for example because it is replicated across all nodes, the Interpretation Handler can simply query its own, local copy. Alternatively, nodes could be allowed to query the broker network for specific interpretation contexts. Both approaches require a naming scheme to identify interpretation contexts uniquely. We suggest a hierarchical naming scheme as commonly used in OOP.

Transforming in the Middleware

We suggest applying the transformations in the middleware (i.e., on broker nodes) rather than in the clients for several reasons:

- **Manageability** of interpretation context changes and supporting new producers or consumers
- **Reusability** of existing interpretation context definitions
- Support for **resource-constrained clients** that have limited processing capabilities
- **Easy integration** of our approach into existing software infrastructures

It is easier to update interpretation contexts if they are stored in the middleware, rather than each client storing its own transformation instructions. The context repository allows for reusing interpretation contexts and easily defining new contexts, helping both reusability and easy integration. Reusing an interpretation context in our case only requires referencing it by an identifier. Defining a new interpretation context is easy if it can be based on already existing contexts. Easy integration is especially important when software from different vendors is used. EBS are typically heterogeneous systems, in which clients of different vendors can be used for publishing and subscribing to events, e.g., by using the advanced message queuing protocol (AMQP) [205]. Thus, transforming in clients requires adapting various code bases rather than adapting a central code base of the middleware. Furthermore, resource-constrained clients like wireless sensor nodes do not have enough processing capabilities to perform transformations.

4.3.2 Declaring Interpretation Context

To remain independent of a specific programming language, we support interpretation context declarations in the widely-adopted XML. Other languages are possible as well - even in coexistence. Thus, developers do not need to learn new specification languages. To illustrate this, we show how the US interpretation context from Figure 4.6 for Java can be declared in XML:

```
<types>
  <type>com.logistics.us.CustomerProductStatusEvent</type>
</types>
<mappings>
  <mapping from="com.logistics.eu.ProductStatusEvent"
    to="com.logistics.us.CustomerProductStatusEvent"
    dir="outgoing" />
</mappings>
<rules>
  <rule pattern="Position" function="toLatLonCoordinates" />
  <rule pattern="Money" function="toDollars" />
  <rule pattern="ProductStatusEvent.dynamicCost" function="toIdentity" />
</rules>
<functions>
  <function name="toLatLonCoordinates" class="com.logistics.us.conversion.toLatLonCoordinates" />
  <function name="toDollars" class="com.logistics.us.conversion.DollarConverter" />
</functions>
```

Annotations

To allow for intuitive and in-code declaration, our Java prototype supports various Java annotations. Developers may use these annotations to implicitly define interpretation contexts in a familiar way (see Listing 4.1 for an example). To define type mappings, the developers can use the `@MapsTo` annotation, supplying the class name of the mapped class. To specify that a class should be transformed with a specific function, developers may use `@TransformWith`.

These annotations are just another form of expressing transformation rules. To simplify things for developers further, we allow for annotating a class's attributes with units (e.g., `@Unit("USD")`). By analyzing the units given by developers and those that the notification service uses, we can generate the appropriate transformation rules. Thus, developers can simply express their *wish* towards the data.


```

@Context("us")
@MapsTo("com.logistics.eu.ProductStatusEvent")
public class CustomerProductStatusEvent implements Serializable {

    @Unit("latlonPos")
    private LatLonPosition pos;

    @Unit("USD")
    private Money price;

    ...
}

@Context("us")
public class CurrencyFunctions {
    @ConvFunction(sourceUnit="EUR", targetUnit="USD")
    public static Money toDollars(Money value) {
        // look up exchange rate and convert
    }

    ...
}

```

Listing 4.1: Annotations in ACTrESS

Conversion functions are defined by annotating a method with `@ConvFunction`, specifying the source and target unit. ACTrESS remembers all such annotated methods and uses them to convert between attributes of differing units.

In the above example, ACTrESS will realize that, when sending a `ProductStatusEvent`, clients with the `us` context want `pos` to be the unit `latlonPos` and `price` to be in `USD`. Since the consumer's context is `us` the source unit is thus `EUR` (as defined by `ProductStatusEvent`) and the target unit is `USD`. ACTrESS then searches through its repository of conversion functions to find a suitable function for the transformation.

The provided annotations (Table 4.1 provides an overview) are not as expressive as the transformation rules, but cover many typical application scenarios (and are even more intuitive through the use of units).

4.3.3 Transformation in Action

As pointed out in Section 4.1.1, we assume that event notifications contain hierarchically structured data types. Flat data types can easily be emulated as they are simply hierarchical data types with only one nesting level.

Rule Resolution and Validation

Before generating code, we ensure that an interpretation context is sound (i.e., it will not cause runtime type errors) by using a type system [75]. The type system checks the following conditions:

Annotation	Meaning	Arguments
@Context	Labels a class as being part of an interpretation context	id: the context's unique identifier
@MapsTo	Specifies the mapping for type	name: the mapped type's fully qualified name
@TransformWith	Specifies a conversion function for an entire class, instead of relying on unit resolution	name: the conversion function's fully qualified name
@ConvFunction	Labels a method as a conversion function	sourceUnit: the input unit targetUnit: the output unit
@Unit	Assigns a unit to a field in a type	name: the unit's (unique) name

Table 4.1.: Overview of ACTrESS's Java annotations.

Type-checking of Individual Transformation Rules

For every rule $p \triangleright f$ in the interpretation context, the type system validates the pattern p (see Definition 3). Upon successful validation, the system ensures that the type of the argument of function f is a super-type of the type specified by p . E.g., if $f : T_2 \rightarrow T_3$ and $p = T_0.a_1(T_1)$ then the system checks if $T_1 \preceq T_2$ holds.

Resolving Rules

For every event type T specified by an interpretation context, our system identifies and stores the set of all *transforming paths* rooted at T , which we call the *expanded rule set* for T . A transforming path tp describes which function should be applied to a (nested) attribute. It is of the form $\text{tp} = \langle T \cdot T_1 \cdot \dots \cdot T_n, a_1 \cdot \dots \cdot a_n, f \rangle$ (cf. Definition 2). The expanded rule set of an interpretation context is the union of the expanded rule set for each of its types.

We obtain the expanded rule set for a type T by exploring its attributes recursively following breadth and then depth. For each such visited attribute, we compare its traversal path against the patterns of the interpretation context's rules. If multiple patterns match, we resolve conflicts among these according to our precedence rules (see Section 4.1.2).

The existence of a resolved transforming path tp implies that there is no other transforming path tp' which is a prefix of tp . Consequently, there is no other transforming path tp'' for which tp is a prefix (since tp would not be a transforming path due to the first condition). When exploring the attribute space of a type T , we do not explore deeper once at least one matching pattern is found. The conversion function is responsible for dealing with nested attributes.

Type Verification

For every transforming path $\text{tp} = \langle T \dots, f \rangle$, our system verifies that f 's return type is compatible with the type specified by the mapping for the event type T . As pointed out in Section 4.2 there is not necessarily a 1:1 relationship between mappings and rules as this would be undesirable in terms of expressiveness. Thus, we do not check mappings against every rule in an interpretation context but rather against the transforming paths. This allows for default rules (which might resolve to conflicting types), which are intended to be overridden by more attribute-specific rules (which produce correct types).

To avoid repeated searching for interpretation context specifications and transformation rule resolution, we dynamically generate and compile a transformation class for every interpretation context. A generated transformation class is specific to a certain event type and accesses its attributes directly. It calls the conversion functions and writes the results back into the message. Thus, we avoid performing any kind of resolution at runtime and the strong semantics of our transformation rules allow for generating code which does not need to rely on inspection techniques like reflection. We illustrate the efficiency benefits of our approach in Section 4.4.

Figure 4.10 illustrates the sequence of actions in our code generation mechanism. After connecting the producer sends its interpretation context as a message. The plugin intercepts this message and extracts the types from the interpretation context. The classes are loaded by our custom classloader. After this, the plugin triggers the code generation. The generation looks at the rules and mappings (see previous section) and generates transformation code for handling the producer's events. The generated code is compiled at runtime and then loaded by our custom classloader.

The plugin keeps a mapping from each client to its interpretation context. Likewise, each interpretation context keeps a mapping from each event type to the corresponding transformator. Since we use hash functions to obtain the mapping, we can retrieve the right transformator in constant time.

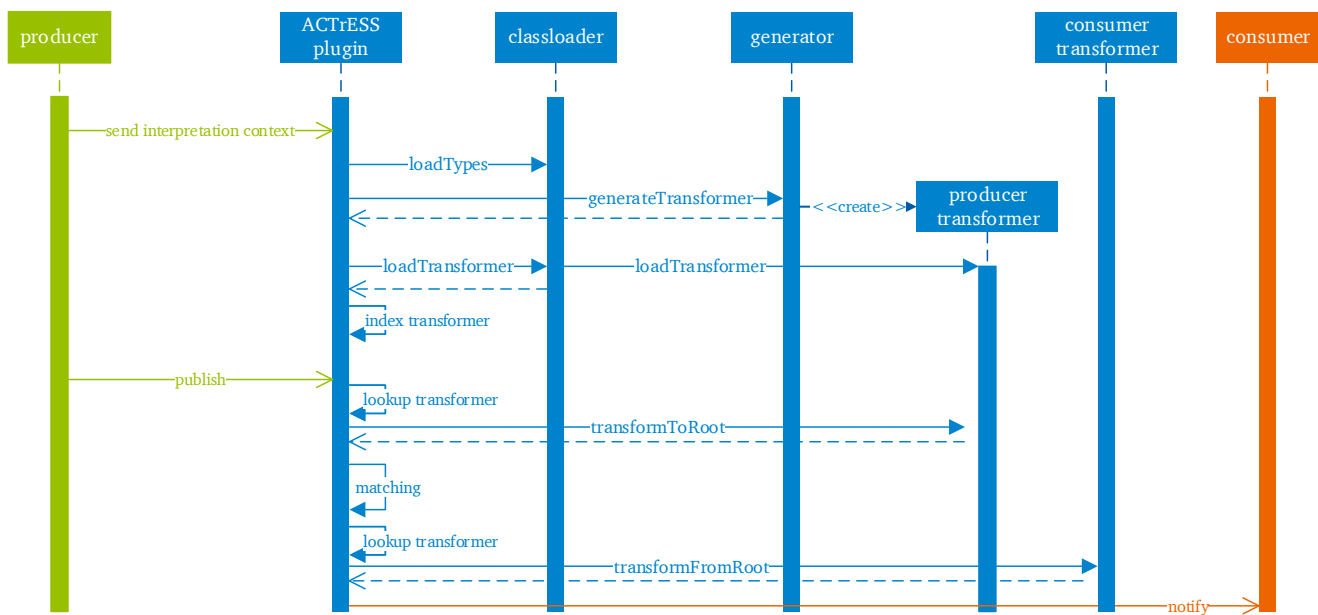


Figure 4.10.: Sequence diagram for the code generation.

When rule resolution and validation or code generation/compilation discovers invalid patterns or type mismatches, it stops and a corresponding error message is sent back to the client. Since interpretation contexts are analyzed separately and only successfully processed contexts can act as a parent, only the error-causing interpretation context will become invalid. In case of an attempted modification, it remains unchanged. All other interpretation contexts remain unaffected and ACTrESS keeps operating with them. We believe that informing the client about an erroneous interpretation context and preventing its use

is better than hoping the invalid rules will not trigger at runtime. Errors in the resolution process are usually indicators of more profound mismatches.

There are cases which do not prevent sound and correct transformations, but indicate specification mistakes. For example, several mappings with identical source type or multiple rules with identical patterns. In both cases we choose the last such mapping or rule and issue a warning to the client. We pick the last rule as we consider this the most up-to-date rule from the user's perspective.

Transformation at Runtime

Upon receiving an event from the producer, the plugin intercepts it and calls the generated and loaded transformer (see Figure 4.10), to transform the message to the root interpretation context. The broker can then do the matching against subscriptions and dispatching of matching events to consumers². Before the event is sent to the consumer however, the broker invokes the consumer's transformer (generation not shown here, as it is similar to the generation of the producer's transformer) to transform the event to the consumer's interpretation.

For any other mechanism than channel-based publish/subscribe, we employ a greedy transformation approach: messages are transformed by the first broker they reach. The message is then routed through the broker network and only transformed to the consumer's interpretation by the last broker in the chain. This is necessary, since subscription and message must share the same interpretation for successful routing. In channel-based publish/subscribe, we can use the following optimization: messages are transformed by the last broker. This allows for comparing the producer's and consumer's interpretation contexts before any transformation is done: if they are the same, we can skip transformation entirely. In addition, we save marshalling and unmarshalling of the message on the first broker. However, a disadvantage of this optimization is that we put more load on the exit brokers, which makes it less suitable for topologies where brokers have highly different numbers of consumers. Furthermore, we might perform the same transformation (producer's interpretation context → root) at multiple brokers, when this could have been done only once on an upstream broker. However, since marshalling and unmarshalling dwarfs the costs of transformation (see Section 4.4), we still see this as a good optimization.

The transformation result is cached to avoid double work in case of multiple consumers with the same context. Since messages are usually rather small in size and the number of interpretation contexts is relatively small, the cache can be kept purely in memory and is large enough to store all transformation results for each message currently processed by the middleware. This avoids accidental latency increases due to disk accesses. Since each broker knows the number of connected consumers, it knows when a cached result is no longer needed and can free resources, avoiding memory contention: if it delivered the message to all connected consumers, it can discard the cached transformation result.

In the case of JMS durable messages – messages that will be delivered to consumers, once they reconnect – this might lead to clearing the cache too early. However, as our evaluation shows, transformation impact is minimal (see Section 4.4). Since durable messages have to be loaded from a persistent storage, transformation overhead is governed by disk access. Thus, we do not add any significant delivery delay.

Because the transformation happens in the middleware, our approach is transparent for any producer or consumer. However, the bootstrapping, i.e., informing the middleware about the interpretation context, has to be done at some point. We assume that this will be done by event-based administrators so that application developers can write producers and consumers without worrying about interpretation.

² Please note that we omit the broker internal details of matching and dispatching in the figure, as they are irrelevant for our approach.

ActiveMQ is an open source, fast and reliable JMS [66] broker developed by the Apache Software Foundation. In ActiveMQ, event notifications are modeled as messages. Note that our notion of interpretation context is orthogonal to that of `JMSContext` from the recently released JMS 2.0 specification – the latter encapsulates several objects used to represent a communication endpoint [66], albeit without modeling its interpretation. We envision that in future releases of the JMS standard, our notion of an interpretation context can become part of the `JMSContext`.

ActiveMQ passes the SPECjms2007 [191] industry benchmark with very good results³. Thus ActiveMQ is an industry-strength messaging middleware whose stability far exceeds that of most research prototypes. We believe that implementing on top of such a widely used middleware further showcases the generality of our approach, as we do not impose special requirements upon the middleware.

ActiveMQ supports plugins via a callback mechanism, informing each plugin about various broker-internal events (e.g., clients connecting/disconnecting, messages arriving and leaving the broker, etc.). Thus, our plugin can intercept messages before and after processing. This enables us to read and modify the payload of messages before they are processed any further and once again before they are dispatched to their respective consumers.

As pointed out above, clients need some way of sending their interpretation context to the broker. We decided against a dedicated API extension, since we aimed at impacting existing systems in the least way possible. Instead, we use dedicated JMS queues. An interpretation context is serialized as a `JMSObjectMessage`, containing the byte code of the context’s custom types and functions, along with the rules and mappings. Our plugin intercepts all messages sent to these dedicated queues and treats them as interpretation context specifications. These special messages are discarded after processing, avoiding unnecessary processing by the rest of ActiveMQ’s message handling stack.

4 Evaluation

To show the capabilities of our approach and implementation, we ran a performance evaluation. We compare our approach with one that relies on Java reflection to dynamically identify the attributes of incoming event notifications and transform them accordingly. This may seem like an unfair comparison, given the usually slow performance of reflection mechanisms. However, this is how typical library implementations work and one of the benefits of our approach is that it provides the same expressiveness as a library approach but enables type safe, static code generation. In addition, our performance results give a quantitative measure.

Furthermore, we compared the code complexity of the code necessary to encode the transformations to support our claim that ACTrESS is easy to use.

4.4.1 Performance Evaluation

We present our performance evaluation results in this section. We first describe the setup we used for the measurements, followed by a description of the scenarios we compared. We then discuss the individual results in detail and give a brief conclusion regarding the performance evaluation.

³ <https://www.spec.org/jms2007/results/jms2007.html>

Setup

Since we are not interested in the performance of the broker network itself, but rather the added overhead, we did measurements with a single broker node. This is a feasible setup, since message traffic can be assumed to be uniform across all brokers. We used a server with Intel Xeon Quad-Core processors with 2.33 GHz and 16 GiB RAM. The workload was generated on a server with Intel Xeon Dual-Core processors with 2.4 GHz and 16 GB RAM, running multiple producers and consumers (see Figure 4.11). Since each producer and consumer establishes its own connection to the broker, it is irrelevant from the broker's point of view that they all run on the same machine. We ensured that the workload generator machine did not become the limiting factor in any way.

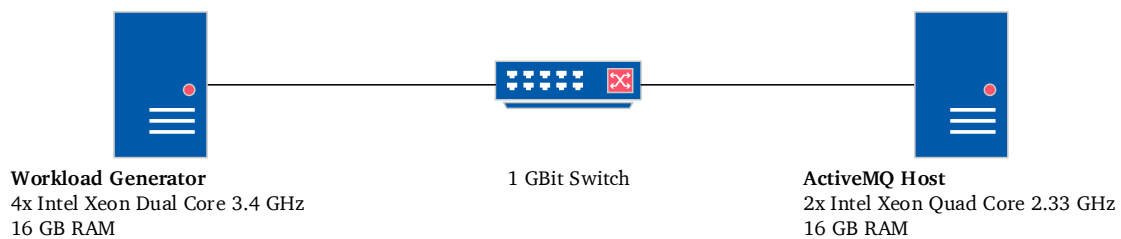


Figure 4.11.: Performance setup

In addition, we also ran a few experiments in a local setting: producers, consumers and a broker instance ran on the same machine. We use this setting to exclude network contention and network latency to gain a better understanding about limiting factors.

We used our experience from DynamoPLV (see Section 1.4.1) and with the SPECjms2007 benchmark [191] to achieve meaningful benchmarking as well as realistic workloads in terms of event size and content.

Scenarios

We compared five scenarios: *baseline*, *content-based*, *actress*, *reflection* and *camel*. *Baseline* simulates channel-based routing: messages are forwarded without accessing the message content, illustrating the smallest latency possible. In *content-based* we simulate content-based routing by accessing message content, requiring unmarshalling of the message. Scenario *actress* uses our approach to transform incoming notifications to the root context and further to the respective consumer contexts. Scenario *reflection* uses reflection for transformation identification and application as a typical competing approach like self-describing messages would do. Finally, scenario *camel* uses Apache Camel for transformations to investigate the impact of using EAI frameworks. EAI frameworks share the idea of integration with our approach, but do not provide implementation details (see Section 6.1 for details). We use compiled transformation classes for this approach, like those that our approach generates. To obtain fair results, we applied the same optimizations (e.g., caching) to all three transforming scenarios. The transformations comprised typical transformation operations like replacing a type and converting between units.

We did not evaluate scenarios in which one interpretation context specializes another one, because specialization happens on a purely conceptual level and – as explained – results in a compiled class (see Section 4.2.1). Thus, it does not matter during runtime how deep in an inheritance-chain an interpretation context was defined.

Results

In this section, we present the quantitative results from various performance measurements.

Raw performance

As a first experiment, we wanted to determine the raw performance before further dissecting the results. To do so, we created a workload that simulates that of SPECjms2007. We could not apply SPECjms2007 directly, because transformed messages cause SPECjms2007's integrity checks to fail which leads to the test being aborted. Thus, we manually created workload classes from the specification (see Listing 4.2). SPECjms2007 simulates complex supply chains between distribution centers, headquarters, and suppliers interact via various inter-company event notifications [173]. We create various producers and consumers to emulate the interaction specified by the benchmark.

```
public class ShipInfo implements Serializable {
    private String shipmentID;
    private String DCID;
    private Date shipmentDateTime;
    private String orderID;
    private String SMID;
    private Address deliveryAddress;
    private String contact;
    private String shipper;
    private List<ShipInfoLine> shipInfoLines;

    // constructor and getters/setters
}
```

Listing 4.2: Message class created based on the documentation of SPECjms2007

Figure 4.12 shows the latency for different notification rates. It is important to note that the vertical axis has a logarithmic scale. As the figure shows, *reflection* adds significant latency overhead compared to *actress*. This is an indicator for the increased computational effort of the reflection-based approach. Due to the increased effort, ActiveMQ cannot cope with higher event production rates, events are stalled, accounting for the steep latency increase. The *reflection*-curve shows a typical curve for a system that gets closer and closer to full utilization, with exponentially increasing latencies. *Camel* performs even worse, because it is not integrated into the broker and thus additional event marshalling and unmarshalling has to occur. This shows that EAI frameworks also suffer from poor efficiency. This is especially interesting, because we used compiled Java code without any reflection for these transformations. Scenarios *content-based* and *actress* achieve more than double the throughput than *reflection* and have no measurable difference, while *baseline* achieves even slightly more events per second.

Because there is no measurable difference between *content-based* and *actress* and because our setup already resembles the worst case where every event has to be transformed (no optimizations due to matching interpretation contexts), we do not provide more details like the effect of the number of contexts. A single context incurs a memory footprint of less than one KB, unless it defines hundreds of types or conversion functions. Thus, a single megabyte of memory can hold about thousand contexts. Furthermore, we omit the results for *camel* in the following evaluations, since its performance is already clearly worse than simply using reflection.

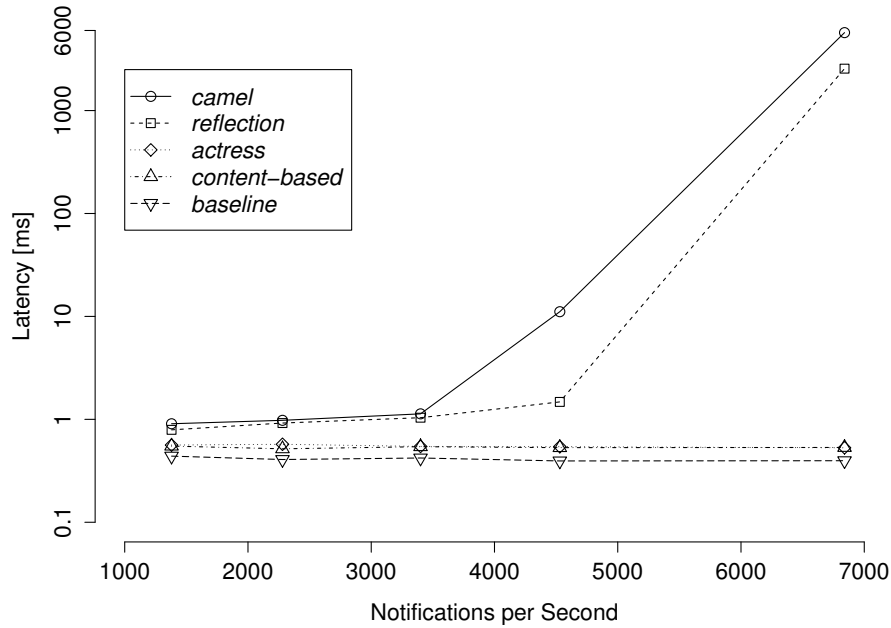


Figure 4.12.: Throughput/latency for the SPECjms2007-inspired workload.

Producer/Consumer Ratio

We evaluated each of the four scenarios with a producer-to-consumer ratio of up to 1:10. We found that this is a realistic ratio in large scale enterprise applications.

Figure 4.13a shows the latencies between the different scenarios with varying numbers of consumers. As the figure illustrates, there is again no measurable difference between *content-based* and *actress*. Compared to the *baseline*, the latency was 20% (one consumer) up to 60% (ten consumers) higher. The reflection-based approach however does not only add to latency, but latency increases much more with a growing number of consumers. Thus, our approach performs and scales better than a reflection-based approach.

We ran the same experiment again in a local setting, where producers, consumers and broker run on the same machine, to make sure that there are no network-related maskings. Figure 4.13b shows the results for this evaluation. Generally, maximum throughput is lower than in a distributed environment, because one machine has to do all the work now. However, the relative throughput performance between the scenarios is the same as above. It becomes even more evident that the reflection-based approach adds severely to latency, as it is ≈ 10 times slower, while ACTrESS does not add any measurable overhead compared to *content-based* publish/subscribe. Differences to the *baseline* were about the same, but are less visible due to the scale of the y-axis.

Besides illustrating that our approach is faster than a reflection-based library implementation, this analysis also shows that our approach is as effective as a tedious manual coding of transformations at clients, since we do not add any measurable overhead to the performance of the pure *content-based* mechanism without any transformations.

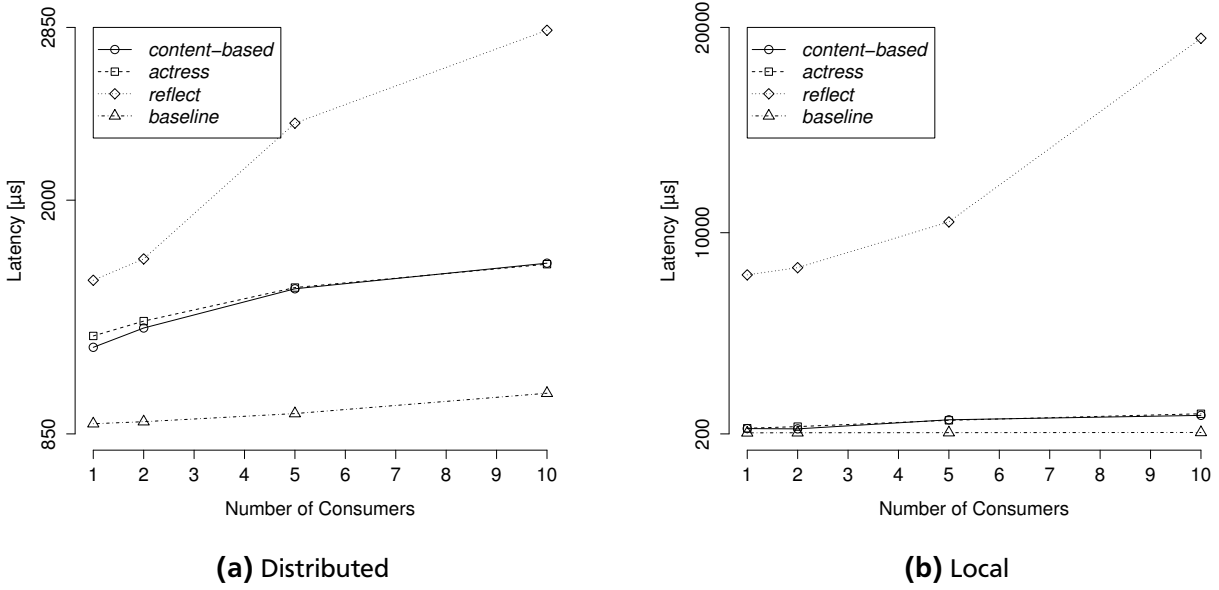


Figure 4.13.: Producer/Consumer ratio

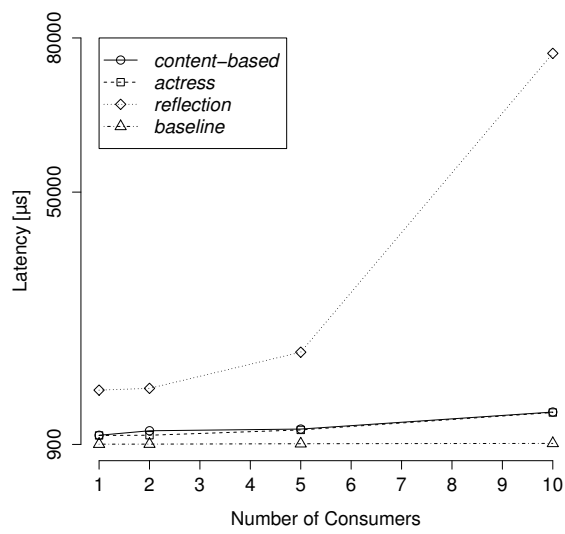
Event Type Structure

Since event notifications usually differ in their complexity, we evaluated the influence of the number of attributes in an event notification. We created event types with a total of 15 attributes, modeling event type sizes at the large end of the spectrum. We structured these attributes in a flat and a deep manner. Flat means that all attributes were on the first nesting level, whereas in deep we created 15 nesting levels.

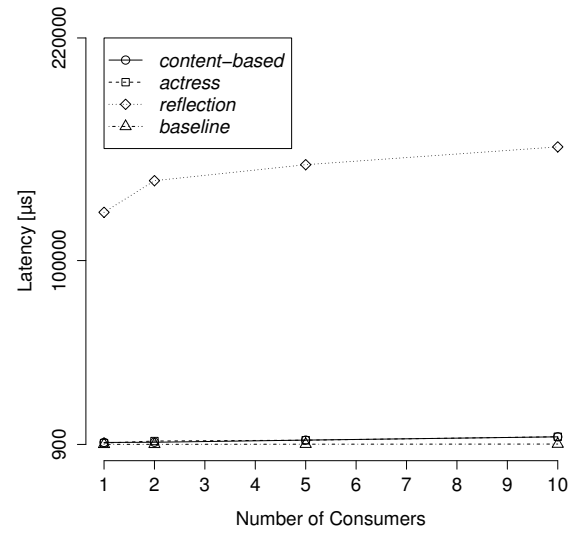
Figure 4.14 shows the impact on latency under the respective modifications. Due to increased serialization and deserialization effort, more attributes generally result in higher latencies. We observed that flat event notifications (Figure 4.14b) have less negative impact on performance than deep structures (Figure 4.14a; please note the different scale on the y-axis). Interestingly, scenarios *content-based* and *actress* perform better with flat structures compared to deep ones, while reflection performs better with a deep nesting level compared to flat structures (the steep increase in the graph does not continue for higher consumer counts). This is probably because expensive accesses are partly amortized over several nested attributes. The difference *content-based* and *actress* is very small, while the reflection-based approach is heavily affected by event structure. In our experience, event notifications yield mostly flat structures and thus the advantage of our approach is even more apparent.

Locality

Since our approach allows for pushing transformations into the middleware and thus closer to the producers, we can avoid redundant transformations. To evaluate the benefit of this, we evaluated the effect of transforming messages close to the producer and the same transformation happening at each consumer. Each measurement used the same amount of resources for producers, broker and consumers. As the results show, being able to push the transformation close to the producer leads to huge performance gains, both in terms of maximum throughput (Figure 4.15a) and latency (Figure 4.15b).

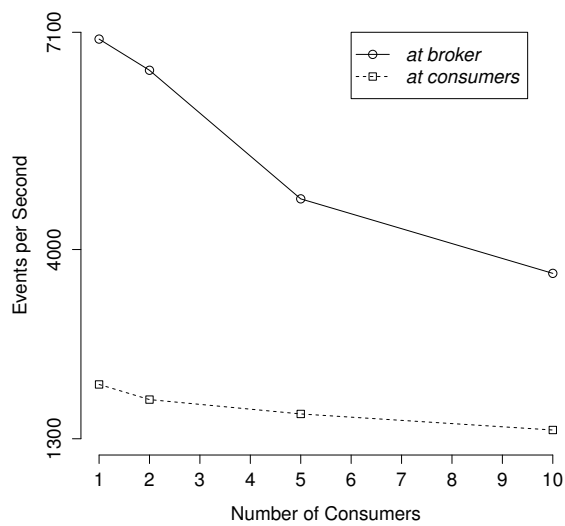


(a) Deep

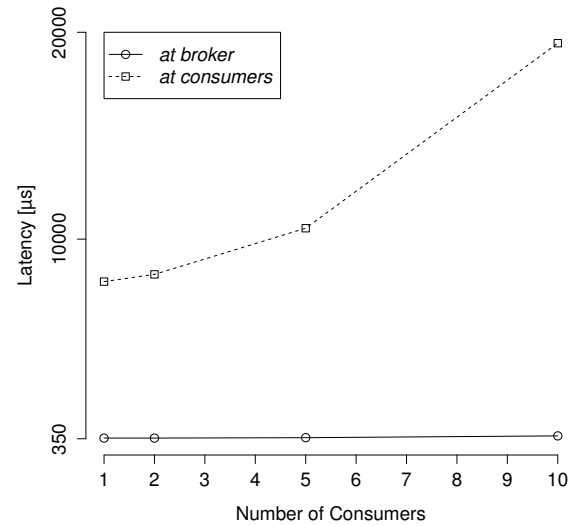


(b) Flat

Figure 4.14.: Impact of Event Type Structure



(a) Throughput



(b) Latency

Figure 4.15.: Impact of Locality of the Transformation

Rule Resolution and Code Generation Overhead

Our implementation generates transformation code for new or changed interpretation context. While we believe that compared to the actual event notification rates, new interpretation contexts or changes to existing ones are rare, we are still interested in the overhead of this step. It is important that this step has acceptable overhead so that deployment and testing can be done quickly and – even more importantly – changes at runtime take as little time as possible.

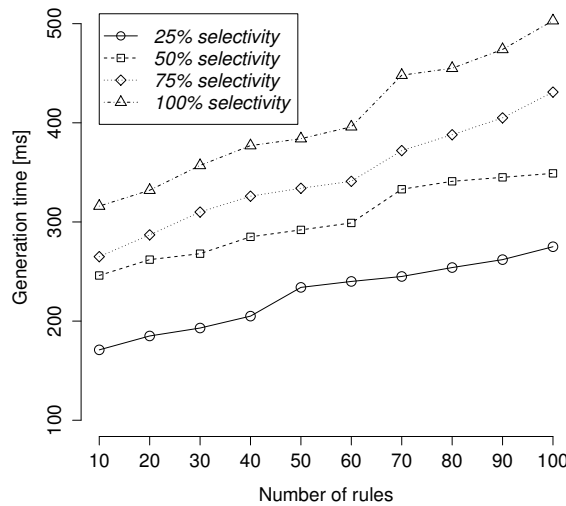


Figure 4.16.: Generation time

Figure 4.16 shows the time it takes to generate 1000 transformation classes for different sizes of the interpretation context's rule set and different selectivity of the rules. Selectivity means how many attributes of the types are actually affected by the rule. We used 1000 iterations to keep the variance low. Our implementation scales linearly with the number of rules and affected attributes. Every rule has to be checked because there might always be a more specific one at the end, and thus this is the optimal result. Similarly, for each affected attribute we have to generate some code. Thus, a generator must have at least linear complexity in terms of number of rules and attribute selectivity.

Figure 4.17 illustrates the impact of a change to the interpretation context (or the a new definition) for the client(s) using this interpretation context. The update is injected into the systems after five seconds. The figure shows the brief increase in latency at this time. Latency increases because messages sent after the update have to wait until the new transformation class is available. This takes a small amount of time due to rule resolution, code generation, compilation and loading (cf. Figure 4.10). After this short delay, the system performs as before. This demonstrates that our implementation can handle changes at runtime without significant impact on performance. Please also note that most of this delay is due to Java internal mechanisms (compilation, loading), since the generation time even for 100 rules and 100% selectivity is less than 1 ms (for one generation, cf. Figure 4.16).

Performance Conclusion

We conclude that our approach is superior to reflection-based approaches. It increases latency of pure channel-based systems without content-based filtering, but does not add any measurable overhead to content-based mechanisms, as shown by the *content-based* scenario. Thus, we can relieve clients from

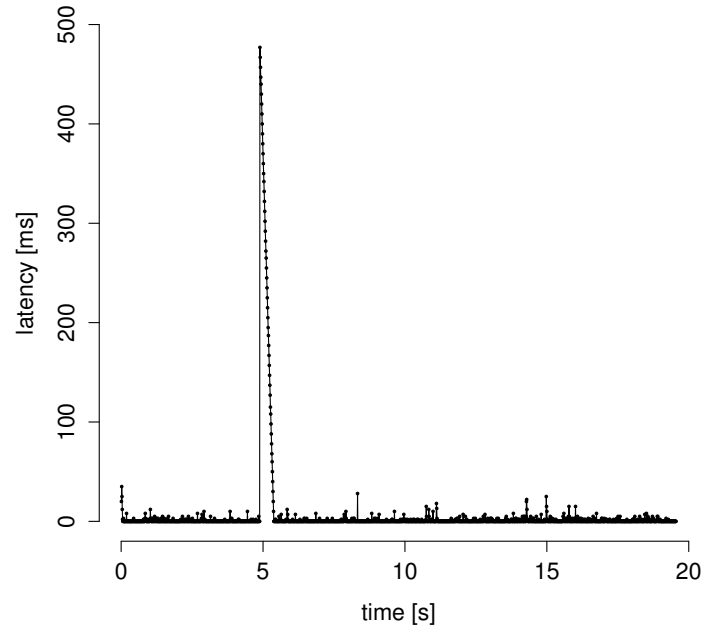


Figure 4.17.: Recompilation overhead

doing transformations manually, which is both tedious for programmers and can negatively affect performance.

Because the added overhead is so little, we omit comparisons with other techniques like message filters using XSL Transformations (XSLT). The clear advantage of our approach is that transformation instructions are very lightweight and easily definable, while the model underlying our approach is provably type-safe [75].

4.4.2 Ease of Use

In this section, we demonstrate the efficiency for the programmers by showing that our interpretation contexts lead to considerably lower implementation effort than manual coding of transformations, and that existing transformations can be changed more easily. We use five typical event-based applications to compare the required lines of code and ease of changes. Kemerer [120] analyzed over 30 software complexity metrics and concluded, that "a number of more complex metrics may be essentially measuring the size of the program or other component under investigation, and therefore may provide little additional information". We thus see lines of code as a valid indicator for code complexity and maintenance effort. Consequently, we use it for our comparison.

To give a brief comparison between the two approaches, consider the example to transform attributes of type `Address`. Our approach needs just one rule to specify the transformation for every occurrence of an attribute of type `Address` in any type. When coding transformations manually, the developer has to write a dedicated if-else branch for every event type with an address attribute at any (!) nesting level. Inside, several lines of code for attribute extraction, object creation and transformations are needed. Listing 4.3 showcases this effort. Even though the transformation is the same every time, it still must be encoded for every event type yielding `Address` attributes.

```

public class ManualTransformer implements Transformer {
    ...
    ConversionFunction addressCF = new ConversionFunctions.get("europeanaddress", "usaddress");
    ...

    @Override
    public Object transform(Object object) {
        if (object instanceof CallForOffers) {
            CallForOffers o = (CallForOffers) object;
            USCallForOffers result = new USCallForOffers();
            result.setDeliveryAddress((USAddress) addressCF.convert(o.getDeliveryAddress()));
            return result;
        } else if (object instanceof Offer) {
            Offer o = (Offer) object;
            USOffer result = new USOffer();
            result.setSupplierAddress((USAddress) addressCF.convert(o.getSupplierAddress()));
            result.setDeliveryAddress((USAddress) addressCF.convert(o.getDeliveryAddress()));
            ...
            return result;
        } else if (object instanceof POrderConf) {
            POrderConf o = (POrderConf) object;
            USPOrderConf result = new USPOrderConf();
            result.setSupplierAddress((USAddress) addressCF.convert(o.getSupplierAddress()));
            result.setDeliveryAddress((USAddress) addressCF.convert(o.getDeliveryAddress()));
            result.setInvoiceAddress((USAddress) addressCF.convert(o.getInvoiceAddress()));
            ...
            return result;
        } else if ...
    }
}

```

Listing 4.3: Every occurrence of Address must be encoded

For our comparison we use the event types specified by various applications:

The SPECjms2007 benchmark We introduced the benchmark in Section 4.4.1. Transformations include address translations between different regional formats and changing product descriptions (in orders, invoices, etc.).

Marketcetera The MARKETCETERA⁴ automated trading platform defines various event types capturing stock ticker quotes and allows for elaborate automated trading. The transformations on this platform perform currency conversions, timestamp formatting changes and renaming of some indicators, assuming differing terminologies being used by the consumers.

HTM The Highway Traffic Monitoring (HTM) system handles data from sensors and cameras along streets and highways, monitoring road, traffic and weather conditions [181]. Such systems are used by many large cities. Transformations include coordinate translation, timestamp format changes and unit conversion.

DRADEL is an application environment for modeling and analyzing distributed architectures, including code generation [137]. Since it runs on top of a message-oriented middleware, multi-user support is possible. In such a setting however, path references and line numbers need

⁴ <http://www.marketcetera.com>

to be adapted to each platform. Thus, operational events of DRADEL need to be transformed accordingly.

ERS The Emergency Response System (ERS) is a distributed application running on multiple mobile devices and helps organizing human resources during natural disasters [155]. Its events often refer to geographical regions, which need transformation between individual users to adapt to their specific format.

Table 4.2 compares the lines of code needed for *specifying* transformations, *changing* existing transformations and *adding* new event types to the system.

	<i>manual</i>			<i>ACTrESS</i>		
	<i>specify</i>	<i>change</i>	<i>add</i>	<i>specify</i>	<i>change</i>	<i>add</i>
<i>SPECjms2007</i>	167	15	92	22	2	1
<i>marketcetera</i>	108	9	37	16	2	1
<i>HTM</i>	237	16	133	21	2	1
<i>DRADEL</i>	417	16	139	30	3	2
<i>ERS</i>	351	12	117	21	3	2

Table 4.2.: Code complexity comparison

As the table shows, ACTrESS requires significantly fewer lines of code to *specify* the transformations in the individual scenarios. This is because transformations are specified by high-level rules based on the semantics of the data, not based on where these semantics appear (see above). We did not count lines of code that can be generated by standard IDEs. The numbers indicate the effort necessary to specify the transformations of one client that needs to transform events. With additional clients needing different transformations, the specification effort has to be made again, multiplying our benefits.

The *change* columns show the number of *individual lines* of code that needed to be changed when a certain type (e.g., *Address*) should be changed into a different format. Again, ACTrESS benefits from its lightweight and intuitive rule language, usually requiring only the adaption of a rule and a mapping.

Finally, the *add* column shows the lines of code which need to be analyzed when introducing a new event type to the system. This can become complicated. Suppose we want to add a sensor to the traffic management system to detect oil on the road (raising an *Oil*) event. Every sensor event in the system has an attribute *SensorMetadata*, which has an attribute of type *Location*. *Locations* need to be transformed between different clients. In the manual approach, it is thus not immediately apparent that adding the *Oil* event requires a new piece of transformation code (for *Location*). The developer has to analyze the existing transformation code and realize that locations are transformed and then write the new code. With ACTrESS, only the mapping has to be defined. As the numbers show, this leads to significantly less effort for adding new types.

Contrary to intuition, EAI frameworks like Apache Camel do not reduce complexity for these steps. Although supporting event transformations architecturally, transformations still need to be coded manually, resulting in the effort depicted above.

Complexity Benefits for Transforming in the Middleware

The advantage of our approach and of transforming in the middleware becomes even more apparent if we consider what happens when a new producer joins the system. If we use a manual approach in

which each consumer is responsible for the transformation, we have to modify (and recompile) each consumer. One might be able to determine that only a certain subset will receive the new producer's event notifications and thus reduce the number of components that have to be modified. However, this does not follow the spirit of event-based systems as every producer might start producing new data and the consumers who were not modified are now unable to understand that producer's event notifications. In addition, a manual approach violates the independence of producers and consumers by forcing a new producer to notify consumers to change.

With ACTrESS, the producer can seamlessly blend into the new system without consumers ever noticing the new participant (except for the published data). This supports the key requirement of anonymity between components.

5 Policy-driven Contextualization

*Programs must be written for people to read,
and only incidentally for machines to execute.*

–Harold Abelson

In the introduction (Chapter 1) we established that heterogeneity and contextualization are two prominent challenges in Cyber-physical Systems (CPSs). After showcasing our approach for resolving heterogeneity in the previous chapter, we introduce and evaluate our contextualization approach in this chapter. Our approach uses high-level, declarative policies for describing situations. It deals with context about events (cf. Section 3.2.2). We will also show how our approach for resolving heterogeneity –ACTrESS– helps with this contextualization approach.

We advocate an approach that introduces the abstraction of a policy. A policy represents an intuitive description of a situation, similar to the idea in [19], and the desired reaction upon detection. It is the conceptual representation of a set of fine-grained rules, which cooperate towards a common goal. Thereby, policies enable abstracting from rule dependencies and management. By allowing users to state policies declaratively, we enable them to focus on what they want to achieve, rather than how this is done exactly. Other approaches, sharing the goal of simplifying the use of CPS by providing higher-level abstractions, focus on procedural workflows [106, 10]. Declarative statements bear the advantage of abstracting from implementation details, while procedural statements provide more control of how a task is achieved. No concept is inherently superior over the other.

To achieve our goal, we provide a middleware approach which allows for defining our example policy of making sure guests are accompanied by employees, introduced in Section 1.2, in an intuitive and declarative way:

```
IF
  person A with property status='guest' IS INSIDE
  room R with property security='restricted' AND
  person B with property status='employee' IS NOT WITHIN 5m of A OR B IS NOT INSIDE R
THEN
  sound alarm
```

Listing 5.1: Example policy for our running example

Providing such a middleware requires the following components (Figure 5.1 provides an illustration):

- A user-friendly way of stating the policies. We refer to this as the *Declarative Policy Language (DPL)*.
- A *grammar* underlying DPL as a formal basis for parsing and interpreting policies.
- A mapping from DPL to *executable code*. We divide the overall functionality into small, lightweight, independent, collaborating parts, which we call *Event Enrichment Components (EECs)*. EECs are automatically generated and can be instantiated in a highly distributed way, taking advantage of scaling capabilities of modern cloud environments. EECs enforce policies by evaluating derived rules against incoming events.

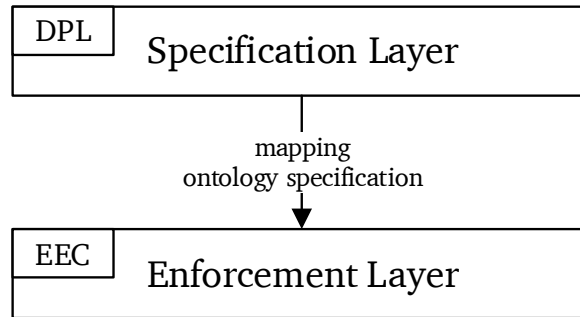


Figure 5.1.: Policies are specified in Declarative Policy Language (DPL) and mapped to executable code in Event Enrichment Components (EECs).

- d) A framework to capture domain concepts in an *ontology* to support the mapping. To generate EECs for the policy given in Listing 5.1, the middleware requires knowledge about functions like **IS INSIDE**, what properties persons have and where to obtain values for them. In our approach this step involves specifying an ontology and annotating data sources with metadata. We use this knowledge in our contextualization step for bridging the gap between data produced and information needed [72].

We will detail each of these components in this chapter. In Section 5.1 we discuss the necessary elements for the ontology component including a critical analysis of alternatives and given the requirements established in Section 3.2.2. Section 5.2 provides details for DPL and its grammar. Section 5.3 gives details on our code generation mechanisms, including an explanation how we efficiently handle NOT and approaches for other challenging problems faced. Sections 5.4 and 5.5 provide information on the middleware architecture and implementation. We evaluate our approach in Section 5.6 both in terms of performance and usability.

5 Ontology Model

Since our goal is to create middleware which is not bound to a specific use case, we need to have some means of capturing domain concepts. This step is similar to creating domain models in software engineering or an Entity-Relationship-Model in database design. The question is which model to use for representing the knowledge of domain concepts in our contextualization approach.

In computer science, ontologies are the digital representation of a piece of knowledge. In recent years, many ontology languages like the Web Ontology Language (OWL) [136] emerged. With a suitable ontology we can let the middleware reason about certain domain concepts, e.g., if a person is inside a room. However, the various ontology languages exhibit different levels of expressive power. The reason not to simply use the language with the highest expressive power is decidability and performance. A higher expressive power usually results in a more complex and hence slower reasoning process. Since we are designing an event-driven middleware, performance is an important factor.

To find an appropriate ontology system, we looked at context-aware systems to derive basic requirements (see Section 3.2.2) and came up with a basic model. In a second step, we validated this model against Event-driven Process Chains (EPCs) provided by Software AG, which they use in their consulting jobs and thus contains frequently updated, typical representations of business processes across a broad range of companies. The remainder of this section describes each step in further detail.

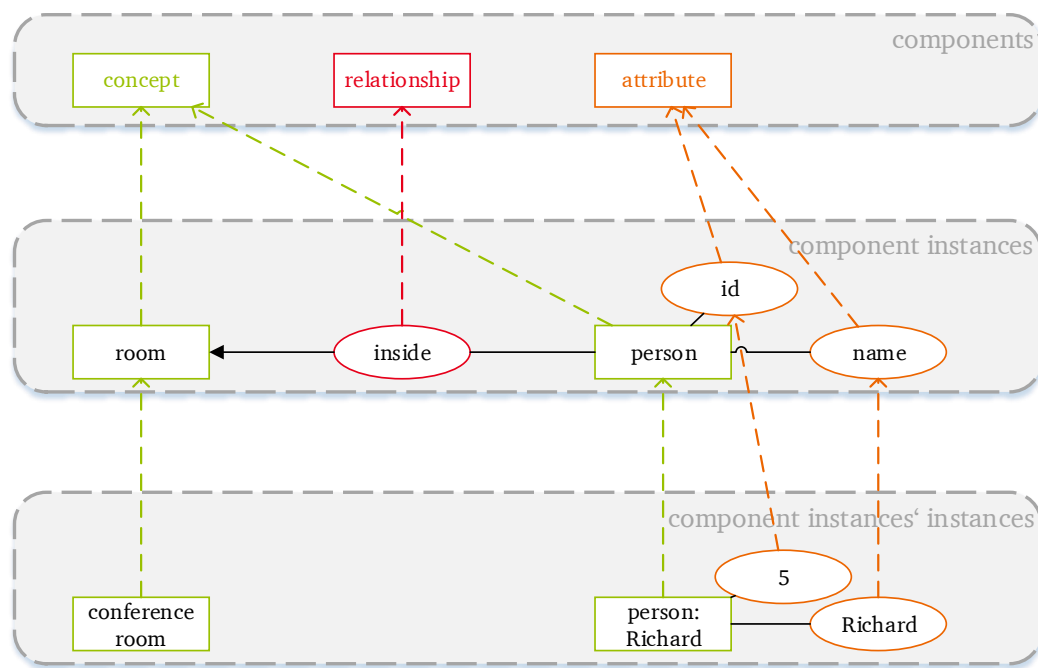


Figure 5.2.: Conceptual view of our ontology model

5.1.1 Model Choice

In this section, we present our ontology model that satisfies the requirements we established in Section 3.2.2 and explore alternatives, arguing why we did not ultimately choose them.

Ontology Model

Our ontology model consists of **three components: concepts, attributes and relationships**, which we explain in more detail in this section. In a concrete deployment of our middleware, each of these components is instantiated (e.g., **room is an instance of concept**). Our middleware can then reason about a particular concept's instances. For example 'conference room' is an instance of **room** and thus a concept instance's instance. **Our ontology model thus forms a metamodel**. Figure 5.2 illustrates this. Note that relationship instances (e.g., **inside**) do not have instances, since a relationship universally describes the connection between concepts and thus all potential instances are the same.

Concepts

Every policy will necessarily refer to certain *things*. In the example from Section 1 these things are **person** and **room**. In that regard, they are similar to entities in database schemata. However, we chose to name these things *concepts* to avoid confusion. Concepts do not need to have a representation in a database or anywhere else. **They might exist implicitly only, e.g., as part of an event's data**. For example, event data about the velocity and acceleration of a football refers to the concept **ball**, without the football being represented anywhere as a dedicated entity.

Attributes

Concepts have attributes which define their properties and may be used to distinguish one concept instance from another (for example, a person's name). Thus, attributes in our ontology model are



After explaining the two components, we define them more formally:

Definition 7 (Concept). A concept C is a 2-tuple $\langle c, \mathcal{A} \rangle$. c is the concept's unique identifier and \mathcal{A} is an unordered set of attributes which belong to C . The function $\text{attrs} : C \rightarrow \mathcal{A}$ returns a concept's attributes: $\text{attrs}(C = \langle c, \mathcal{A} \rangle) := \mathcal{A}$ and $\text{attr} : C, a \rightarrow A$ returns a concrete attribute identified by a : $\text{attr}(C = \langle c, \mathcal{A} \rangle, a') := A : A.a = a'$.

Relationships connect concepts with an assigned meaning. They are, however, different from relationships in ERM: while a relationship in the ERM is property of the entity instance’s primary key and thus relatively static, relationships in our model are subject to any of their attributes. Our model also includes a pointer to how to determine if a relationship between two concrete concept instances holds. This is done by referencing a *relationship evaluation function*.

It is possible that two concepts are connected by more than one relationship. For example, the concepts employee and room can be connected by the relationship works in, indicating their office(s) and meets customers in, indicating a meeting room with customers.

Relationship evaluation functions evaluate whether a relationship between two instances of concept instances (e.g., Richard and the conference room) holds. Since relationship functions can be complex, we chose to keep them separate from the ontology definition itself. This is in line with the common principle to keep functions and structure separate to support *separation of concerns*. In our example, a structural concern is to say that generally, persons can be inside rooms. A functional concern is to say what exactly the semantics of being inside a room is. Thus, a relationship evaluation function is a function that maps two attributes to a boolean value:

Definition 8 (Relationship Evaluation Function). A relationship evaluation function is a function $f : T, T \rightarrow \{\top, \perp\}$. The function $\text{typeof} : F, i \rightarrow T$ returns the type of the i th parameter of f .

For example, `inside` refers to a relationship evaluation function

```
boolean insideFunction(Position pos, Coordinates cos) {  
    return (pos.getX() >= cos.getLowerLeftX())  
        && (pos.getX() <= cos.getUpperRightX())  
        && (pos.getY() >= cos.getLowerLeftY())  
        && (pos.getY() <= cos.getUpperRightY());  
}
```

To support relationships like `within`, which need an additional parameter (e.g., 5 meters), the model must support parameterized functions. One way to support this are function objects, that are instantiated at runtime with appropriate parameters.

We can use Definitions 6, 7, 8 to formally define relationships:

Definition 9 (Relationship). A relationship is a 5-tuple $R = \langle C_1, C_2, a_1, a_2, f \rangle$ where C_1 and C_2 are concepts, a_1 and a_2 are attribute identifiers and f is a relationship evaluation function such that $\text{attr}(C_i, a_i) \neq \emptyset : i \in \{1, 2\} \wedge \text{typeof}(\text{attr}(C_i, a_i)) = \text{typeof}(f, i) : i \in \{1, 2\}$.

The three components also form the basic building blocks of policies, as Listing 5.1 illustrates. This model satisfies our requirements. Space can be represented by defining concept attributes and event payload. Time is a property of every event. Since we do not make any assumptions about the data types or structure of the domain model, our model is very flexible. Likewise, attributes and concepts may represent any level of granularity. By using a metamodel oriented approach, we allow for context construction to happen at runtime, that is, concept instances' instances are created at runtime. The structure of the contextual information is constructed at design time which allows for fast reasoning. Relationships provide a basic context reasoning. They can also be used as hooks to support quality monitoring and handling ambiguity. Sensed values can have an additional certainty attribute which the relationship function can respect. Because the three basic components concept, relationship and attribute can be instantiated multiple times, we support multi-context modeling.

Alternative Models

In this section, we discuss alternatives to our custom ontology model. Using already existing, established models should usually be the goal. However, we found that existing models do not fulfill all of our requirements or are not geared towards the properties of Event-based Systems (EBSs) (cf. Chapter 4).

Bolchini et al. used their framework to analyze and classify 16 existing, well-known context models [25]. We compared our requirements (cf. Table 3.2) to their results and found that none of their analyzed models fitted our requirements. The model that is closest to our requirements is the Context Dimension Tree from the Context-ADDICT project [24]. However, it does not support context reasoning or multi-context handling. Other models do not support a variable context granularity or lack in their representation of space and time.

Apart from these models, there is a multitude of ontologies in OWL. We are, however, not interested in a concrete ontology, but rather if its underlying model is suitable. We thus looked at the suitability of OWL-DL, which is the de facto standard for ontology-based context models [19] and would thus be a logical choice. However, the reasoning in OWL-DL is very complex and computationally expensive, as Bettini et al. point out [19]. It is thus unsuitable for EBSs, which rely on fast, low-latency event

dissemination. As the performance analysis for ACTrESS shows, expensive computations on the critical path of events can severely impact performance (see Section 4.4.1).

We agree that it would have been nice to reuse an existing model, but to the best of our knowledge, there is no suitable model which satisfies all of our requirements. In retrospect however, we think that it was a blessing having to develop our own model, because the simple model of concepts, attributes and relationships is very close to the mental model of humans, as the evaluation will show (see Section 5.6) and thus helps with our primary goal: to make CPSs more usable.

5.1.2 Validation Against Industry-ready Event-driven Process Chains (EPCs)

We created our model by analyzing scientific works. To make sure that it is also applicable to real-world scenarios, we validated it against industry requirements in cooperation with Software AG. Software AG uses a reference framework of EPCs for their consulting work. This framework is captured in the *Industry Performance Ready* database. It consists of typical business processes for various company types like logistics, chemical engineering or manufacturing. Jointly with Henriette Röger, we analyzed representative EPCs for their constellations, patterns and semantics. We compared the results with the expressivity of our ontology model [162].

We found that users would benefit from more elaborate structural support. *Generalization and specialization* help in organizing common attributes of concepts, e.g., Accountant and Manager are both Employees with a name and an address. This is similar to Object-oriented Programming (OOP) modeling. *Grouping* helps with forming semantically meaningful groups of concepts and allows for referring to a group as a whole. *Relationships with variable components* are generic relationships, which are not tied to a specific concept, but to a dynamic group of concepts, all of which can form the relationship. Finally, *containers* allow for modeling part-of relationships, with an arbitrary nesting level.

Our analysis further shows that, while supporting these structural items benefits users, they can be expressed with our ontology model. Thus, we decided not to extend the ontology model, as we believe that more complex models hurt performance. Instead, we suggest supporting generalization and specialization, grouping, relationship with variable components and containers in the ontology editor, which transparently maps each structural item to its underlying model counterpart.

5 Policy Language

The policy language serves as the interface to the user. Since our target users are domain experts and not computer experts, a natural language approach may seem as an intuitive choice. However, natural language texts bear ambiguity. Even though humans excel at resolving these ambiguities with contextual knowledge, they often have to ask for clarification nonetheless. We want to avoid these interactions, as it would become tedious and complex to further specify parts of a natural language description which the computer system did not understand. Thus, we believe that the determinism of a formal language outweighs the familiarity of natural language. We designed the *Declarative Policy Language (DPL)*, a formal language in which policies are expressed.

Listing 5.2 shows the grammar that generates DPL in Extended Backus-Naur Form (EBNF) [115]. The terminals concept, attribute, value, f-name, parameter and action are simple strings. Policies are divided into a conditions part and an actions part (similar to Event-Condition-Action (ECA) rules, cf. Section 2.2): the first part contains the triggering conditions for the policy (the situation), while the actions parts lists the actions to take once the policy triggers. The relation of two concepts comprises a condition. We

support conjunction, disjunction and the sequence operator (followed-by). Together with negation, we support the fundamental operators of event algebras [63].

```

policy      = 'concepts', concepts, 'conditions', conditions, 'actions', actions ;
concepts    = concept-def, {concept-def} ;
concept-def = concept, alias, ['(', property, {property}, ')'] ;
property    = attribute, attr-op, value ;
attr-op     = '=' | '!=' | '<' | '>' | '<=' | '>=' ;
conditions  = condition | conditions, op, conditions | '(', conditions, ')' ;
condition   = alias, function, alias ;
function    = ['IS'], ['NOT'], f-name, [parameter], ['of'] ;
op          = 'AND' | 'OR' | 'FOLLOWED-BY' ;
actions     = action, {action} ;

```

Listing 5.2: Grammar of DPL in Extended Backus-Naur Form

Concepts have an alias for referencing within the policy and for managing identity. We want to illustrate the meaning of aliases with the following two partial policies:

```

concepts
  person A with property status='employee'
conditions
  A IS INSIDE ... AND
  A IS FACING ...

```

```

concepts
  person A with property status='employee'
  person B with property status='employee'
conditions
  A IS INSIDE ... AND
  B IS FACING ...

```

The left policy mandates that an employee should be inside a room and facing an object. While the right policy has the same relationship functions, it allows for one employee to be inside the room and a different employee to be facing the object. However, we do not force different aliases to refer to different concept instances, i.e. **A** may refer to the same employee that **B** does. We refer to this as the *identity problem* and elaborate on it more in Section 5.3.3

The property conditions allow for filtering out instances of the concept not relevant for the current condition. For example, guests and employees are both persons, but we only need to be worried about guests being alone in restricted areas. Functions connect two concepts, referenced by their alias, and may have a parameter as indicated in Section 5.1.1. The element 'of' is only syntactic sugar as it makes policies more readable. Our grammar allows for policies to be written using infix style. This makes parsing them slightly harder, but we believe that it is more natural for the user to "think infix", rather than having to write something like `INSIDE(person A, person B)`.

Keen readers will have noticed that our grammar cannot generate the example policy from the beginning of this chapter (Listing 5.1). The reason is simple: while explaining the motivation, we did not want to introduce the details of concepts.

Note that we do not allow NOT in front of operators. For the boolean operators (AND, OR) this is not a problem as any boolean term can be converted to one where NOT operators only appear right before atomic terms. The term NOT (relationship FOLLOWED-BY relationship) however, cannot be directly expressed. We deliberately designed the grammar this way, as we believe that the semantics of such a term are not intuitive. For example,

NOT ((**A** IS INSIDE **R**) FOLLOWED-BY (**B** IS INSIDE **R**))

is also true if (A IS INSIDE R) never happens. It is more likely that the user meant

(A IS INSIDE R) NOT FOLLOWED-BY (B IS INSIDE R)

which is, despite a potential intuitive difference, logically equivalent to

(A IS INSIDE R) FOLLOWED-BY (B IS NOT INSIDE R).

If the user really wants to express the former case, they may use

((A IS INSIDE R) FOLLOWED-BY (B IS NOT INSIDE R)) OR (A IS NOT INSIDE R)

5 Contextualization Mechanisms: From DPL to Executable Code

In the previous sections we explained three of the four necessary components that we mentioned at the beginning of this chapter. In this section, we detail the remaining and most important component: how to get from a policy in DPL to executable code.

To provide details on how to generate executable code that checks a given policy, we decided to illustrate the steps in chronological order. We group the various steps into three phases: startup, policy registration, and operation.

5.3.1 Setting the Stage: The Startup Phase

The startup phase consists of anything that happens before the middleware is ready to process policies.

Loading The Ontology Model

First, the ontology model is loaded. After loading, we check its consistency to avoid hard-to-trace errors later: we verify that every relationship provides the attributes necessary to evaluate the relationship evaluation function and that the expected types of the function match those of the attributes. Furthermore, we check that all referenced attribute classes (e.g., `com.example.Position` for the attribute position) are known to the classloader. Finally, we ensure that relationships provide a reference to their relationship evaluation function and that this function is known to the classloader. If all steps succeed, we proceed with actually loading the classes.

Registering External Sources

External sources provide information about concept attributes. The example from the beginning of this chapter illustrates that there are various data sources contributing information to the overall goal of detecting security violations: RFID scanners and cameras provide location information of persons. Employee and guest data reside in a database while room coordinates may be found in an Extensible Markup Language (XML)-backed floor plan. We distinguish between two categories of data sources: pull sources and push sources.

Pull sources are data sources that need to be queried for information, like databases or XML documents. Push sources are those data sources that actively publish data without an explicit request. These data are

then disseminated to interested consumers in a publish/subscribe fashion. Sensors are a good example of push sources.

We must respect the different interaction paradigms of push and pull sources: pull sources provide comparatively static information, which data consumers actively query from them. Push sources on the other hand provide volatile, high frequency data in an event-based fashion. In the next section, we will see how we handle the two different interaction paradigms. For now, we just want to make the distinction that we call attributes for which pull sources provide data *query attributes* and with associated push sources *event-based attributes*.

Pull Sources

Pull sources need to be annotated with the concepts about which they provide information. For example, the employee table provides information about the concept **person** and needs to be annotated appropriately. Listing 5.3 provides an example of how such an annotation might look like.

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources xmlns="http://dvs.tu-darmstadt.de">
  <sqlsource server="london.dvs.informatik.tu-darmstadt.de" username="..." password="..."
    database="contextualization">
    <table name="persons" concept="Person" keyColumn="id">
      <column columnName="id" attributename="id" />
      <column columnName="firstname" attributename="firstname" />
      <column columnName="lastname" attributename="lastname" />
      <column columnName="type" attributename="status" />
    </table>
  </sqlsource>
  <xmlsource location="building_information.xml">
    <tag xpath="/floorplan/room" concept="Room" key="name">
      <element name="name" attributename="id" />
      <element name="coordinates" attributename="coordinates" />
    </tag>
  </xmlsource>
</datasources>
```

Listing 5.3: Example declaration of external sources

In this example, there are two different kinds of datasources: an SQL source (a relational database, accessed via SQL) and an XML file. The SQL source is basically a mapping of the database's columns to attributes of a concept. Thus, concepts and tables do not need to use the same names. In addition to the mapping, the datasource declaration also contains information about how to connect to the server and which columns uniquely identify a row. Similarly, the XML source defines the mapping using XPath expressions and also contains information about the location and keys.

We use this information to initialize an instance of a Data Access Object (DAO) [196] for each data source. The DAO enables unified access to the datasource, independent of its concrete type. Thus, our middleware can easily be extended to support additional datasources, e.g., web services.

One problem that arises is that information for a single concept (e.g., person) can be spread across various datasources. For example, a person's name and contact information might be stored in one table and the status in another table. This is no problem as long as the sets of attributes the DAOs provide are disjoint (except for the key attributes), because each datasource can be queried independently, even in

parallel. In the general case, however, sets are not disjoint and we are faced with a set cover problem. Given a set of elements \mathcal{U} and a set \mathcal{S} of subsets of \mathcal{U} , the minimal set cover problem is the task of finding a minimal cover $\mathcal{C} \subseteq \mathcal{S}$ whose union is \mathcal{U} . The minimal set cover problem is NP-hard [119]. In our case \mathcal{U} consists of all the concept's attributes and \mathcal{S} are the sets of attributes of each datasource. Luckily, we need to solve the set cover problem only once at startup so runtime performance is not affected. If a pull source is added during operation, we re-compute the set cover in the background and once finished, switch to the new distribution.

There are other challenges for integrating different sources into one system. For example, different schemas, which use synonyms for relation attributes or model associations in a different way [65]. While we are aware of these challenges, we consider solving them out of scope of this thesis.

For performance reasons, it may be beneficial to cache query results. However, efficient caching brings its own set of challenges, especially in environments where more than one data source is involved. Conceptually, our DAOs support caching, but we did not implement a concrete mechanism.

Push Sources

Since our ontology reasons on the level of concept attributes, we require that push sources provide information about a single attribute only (e.g., a person's position) and that this information is in the format defined by the ontology. The attribute may have a complex type (e.g., position consists of at least an x and a y value) and thereby support results from an event-based composition process. Since we are faced with a heterogeneous environment (cf. Chapter 1), we encode the expectations underlying our requirements in an interpretation context (cf. Chapter 4). Thus, we use ACTrESS to transform events to the ontology's format or possibly filter a certain attribute from them.

We also use ACTrESS to transform identity information. We want to illustrate this with our example. Since people are identified by an ID, the middleware will use this id to lookup a person's status in the database (cf. Listing 5.3). However, RFID readers probably do not have access to person ids and instead can only report the tag id. In fact, multiple tag ids might map to the same person id. We can use ACTrESS for transforming the tag id to a person id, before the event is further contextualized.

Please note that while we have put the registration of push sources conceptually in the startup phase, it may happen anytime, even while a policy is already being processed. The new push source will be integrated seamlessly.

5.3.2 Setting Up: Policy Registration

Policies are sent to the middleware as plain text together with a unique identifier. We use the messaging facility of the underlying Message-oriented Middleware (MOM) to do this: policies are sent to a dedicated queue. In addition, we create a command channel for this policy, which the policy components can use to interact. The command channel is registered at the MOM and exists for as long as the policy runs.

Policy Parsing

Upon arrival of a new policy, it is parsed and any eventual errors are returned to the client. We then verify the policy against the provided ontology model to ensure that all concepts and attributes used by the policy are known to the ontology model. Again, any errors are returned to the client.

Especially in multi-user environments, but also with just one user, it might happen that policies conflict. For example, one policy could state to lower the blinds upon the detection of bright, direct sunlight. At the same time, another policy might mandate opening the windows if the oxygen level drops below a certain point. However, lowering the shutters will nearly stop the oxygen flow, despite opened windows. Thus, the first policy's action prevents the second policy from succeeding. It is generally desirable to detect conflicting policies and inform the client that the policy conflicts with existing ones or outright decline the policy. It is, however, extremely challenging to do general policy conflict detection. The challenge lies in understanding the *effect* of an action. As the example illustrates, lowering the shutters has effects which are not immediately obvious. Understanding that these two policies might conflict requires a real-world model which allows a software agent to judge all the effects of a given action. We believe that this requires a lot of semantic annotation of condition actions. However, this kind of semantic analysis is out of the scope of this thesis and partly overlaps with goal conflict detection and resolution in agent-based programming. As a simple alternative, we suggest supporting users by displaying similar policies to them as they create/edit theirs. Similarity can be determined based on the concepts and their attributes a policy is referring to.

Normalization With the Sequence Operator: ECNF

If we treat relationship expressions, e.g., **A inside R** as atomic terms, the conditions part of policies can be seen as a boolean expression. Unlike traditional boolean expressions, however, we have an additional operator: **followed-by** statements act as the sequence operator in event algebras.

We wanted to normalize policy conditions to be able to uniformly process them. Thus, we developed a method to include the sequence operator in the normalization of boolean expressions. We call the resulting normal form the Extended Conjunctive Normal Form (ECNF). ECNF is like the Conjunctive Normal Form (CNF), but allows for **followed-by** (\rightarrow) statements as another level inside disjunctions. For example

$$(A \vee B) \wedge (C \vee D \vee (E \rightarrow F))$$

is in ECNF, while

$$A \rightarrow (B \wedge C)$$

is not.

Conversion to ECNF is done like a standard conversion to CNF with the following modifications to support followed-by statements. Followed-by statements of atomic terms (e.g., **A inside R FOLLOWED-BY A inside Q**) are treated as atomic terms. The pattern **atomic term followed-by disjunction/conjunction** is converted by distributing the atomic term to all terms of the conjunction/disjunction. The pattern **conjunction/disjunction followed-by atomic term** is converted by creating a conjunction/disjunction of followed-by statements. Transitive followed-by statements (e.g., $A \rightarrow B \rightarrow C$) are broken apart with a conjunction. Table 5.1 illustrates this. We can thus convert any expression generated by the DPL grammar (cf. Listing 5.2) to ECNF, for example:

$$\begin{aligned} & A \rightarrow (B \wedge (C \vee D)) \\ \Leftrightarrow & A \rightarrow ((B \wedge C) \vee (B \wedge D)) \\ \Leftrightarrow & A \rightarrow (B \wedge C) \vee A \rightarrow (B \wedge D) \\ \Leftrightarrow & (A \rightarrow B \wedge A \rightarrow C) \vee (A \rightarrow B \wedge A \rightarrow D) \\ \Leftrightarrow & (A \rightarrow B \vee A \rightarrow B) \wedge (A \rightarrow B \vee A \rightarrow D) \wedge (A \rightarrow C \vee A \rightarrow B) \wedge (A \rightarrow C \vee A \rightarrow D) \end{aligned}$$

As the example shows, when converting to ECNF a short term can lead to a comparatively long ECNF term. This is generally true for the conversion to CNF which may produce exponentially many terms and is NP-hard. However, our experience shows that the number of statements in policies is small. Furthermore, the normalization is done offline and only once for every policy, so it will not affect operational performance.

original term	converted term
$A \rightarrow B$	treated as an atomic term
$A \rightarrow (B \wedge C)$	$A \rightarrow B \wedge A \rightarrow C$
$A \rightarrow (B \vee C)$	$A \rightarrow B \vee A \rightarrow C$
$(A \wedge B) \rightarrow C$	$A \rightarrow C \wedge B \rightarrow C$
$(A \vee B) \rightarrow C$	$A \rightarrow C \vee B \rightarrow C$
$A \rightarrow B \rightarrow C$ (associative)	$(A \rightarrow B) \wedge (B \rightarrow C)$

Table 5.1.: Overview of followed-by conversions to reach ECNF

While we developed ECNF to uniformly treat policies, we believe that ECNF has similar benefits as CNF for all areas where a sequence operator is important, for example

- in EBSs, normalization can help to find similar or identical ECA rules;
- in workflows, normalization can help with automatic reasoning;
- as we will see, ECNF limits the maximum height of the policy logic tree (see next section) which in turn puts an upper bound on the processing steps.

Policy Logic Tree

To further process the policy, we transform the ECNF-normalized form into a tree-like structure which organizes logical operators and relationship statements as well as attribute constraints. The policy logic tree for our example policy (cf. Listing 5.1) is given in Figure 5.4.

The leaves represent the concepts together with their alias and their attribute constraints. One level higher are the relationships. The edges connecting relationships and concepts carry the concept's attribute required for evaluating the relationship and an indicator if this attribute is provided by a pull or a push source. Higher up the tree, the relationships are connected by logical operators according to the policy. In the remainder of this section, we provide details how the policy logic tree is constructed.

After ECNF-conversion, we create the policy logic tree from the normalized form. The first step is to create the leaves of the tree. We generate one leaf per alias, including the given constraints. After this, we create nodes for the relationships. Each relationship node keeps references to the relationship evaluation function and its leaves. The edges to the leaves carry the attribute of interest and whether this attribute is an event-based or query attribute. In the next step, we use the conditions in ECNF to connect the created relationship nodes to form the tree. Thus, we get a tree according to Definition 10.

Definition 10 (Policy Logic Tree). A policy logic tree $\mathcal{T} = \langle r, O, F, R, L, E_r, E_o, E_f, E_R \rangle$ consists of a root r , a set of or nodes O , followed-by nodes F , relationship nodes R , and a set of leaves L . The edges E_* connect the nodes, where $E_r = \{(r, n) : n \in O \cup F \cup R\}$, $E_o = \{(o, n) : o \in O \wedge n \in F \cup R\}$, $E_f = \{(f, n) : f \in F \wedge n \in R\}$ and $E_R = \{(r, l, a, p) : r \in R \wedge l \in L \wedge p \in \{\text{event-based}, \text{query}\} \wedge a \text{ is an attribute of the concept of } l\}$

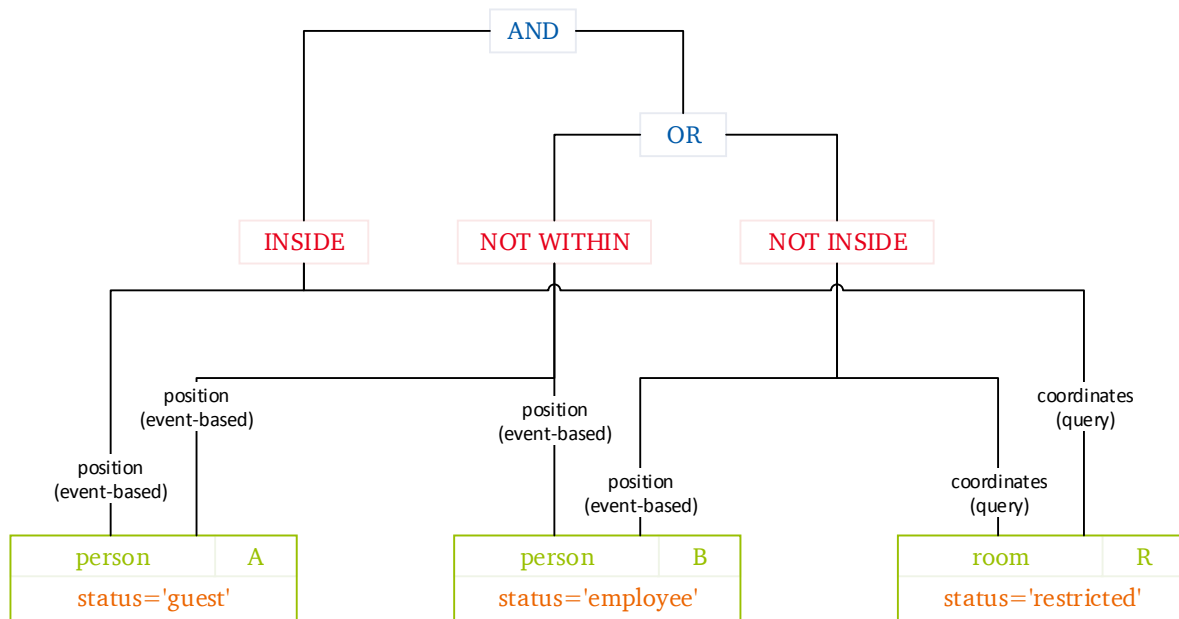


Figure 5.4.: Example policy logic tree

The definition is close to that of a normal tree, but with constraints on which kind of nodes may link to which other kind of nodes. The definition requires that there is a single and-node, the root. If or-nodes are present, they are right below the root node. Followed-by nodes can only have relationship nodes as children, while relationship nodes always have leaves as children.

Limiting Delay: Maximum Tree Height

It follows from Definition 10 and the previous section before that the policy logic tree has a known **maximum height of five**. Since information flows from the leaves to the root, we can guarantee that no matter how complex the original policy is, there will be a maximum number of five hops in the tree. Of course, as the conversion example above illustrates, the tree might get very wide. However, width is not a problem as anything on the same level can be processed in parallel. We thus optimize for total delay at the cost of a potentially higher computation effort.

5.3.3 Setting Things in Motion: Code Generation

After the creation of the policy logic tree, we generate the necessary code for executing the policy. We call the generated components Event Enrichment Components (EECs) and generate one per node in the tree. This allows for distributing the processing of a single policy to many processing nodes, which we will detail further in Section 5.5.

Technically, there is no need for generating code. The generated code is the embodiment of some analysis and specific to event types and attribute types. Thus, its functionality could be mimicked at runtime using reflection. However, as we have already seen in Chapter 4, reflection is slow and resource intensive. Since the enrichment is on the critical path of processing an event, we try to impose as little latency as possible.

Code generation order follows the dataflow: from the leaves to the root. The reason for that is that EECs communicate in an event-based fashion and we need to know the output event types of a node's children, so the EEC for that node can handle them. Together with the generation of the EEC itself, we

also generate an event type to represent the processing of the EEC. We store the format of this event type in the corresponding node. Thus, each node always knows what kind of events to expect from its children.

Preliminaries: The Identity Problem

Aliases provide a form of identity for concepts when referring to concepts in policies. While an alias does not refer to a specific concept instance's instance (e.g., the person with id=5), it binds a detected identity to all of its occurrences. Our example policies illustrates this: If the guest Peter (id=5) is detected in the restricted room R123, A is bound to 5 and R to R123 for all other parts of the policy. Thus, an employee in room R217 does not affect the outcome of this policy instance.

Leaves

Earlier we said that we generate an EEC per node, which is a little inaccurate for leaves. Since policies are defined for EBSs, their processing is triggered by events. Events, however, do not refer to concepts, but an attribute of a concept (cf. Section 5.3.1). Furthermore, we only want to react to those events which are relevant for the policy. Relevant events are those that contain information about an attribute for one of the policy's relationships. Consequently, in this step we generate an EEC per leaf-relationship edge with an event-based attribute ($\{e_R = (r, l, a, p) \in E_R : p = \text{event-based}\}$). We assume that event-based attributes change much more frequently than their query counterparts.

This means that policy processing is triggered when events of push attributes are sent. It also means that a policy without at least one relationship referring to an event-based attribute is never processed. We do believe that it is a valid assumption that at least one such relationship exists as otherwise a policy would be just a query.

Each such generated EEC reacts on events about the event-based attribute. It gets the event type from the ontology model. It uses the id information to query any constraint attributes from the pull sources. If it finds an instance satisfying the constraints, it generates a new event (cf. Figure 5.5). The new event is identical in structure to the input event, but of a new event type, to represent the semantics of satisfied constraints (cf. Listing 5.4).

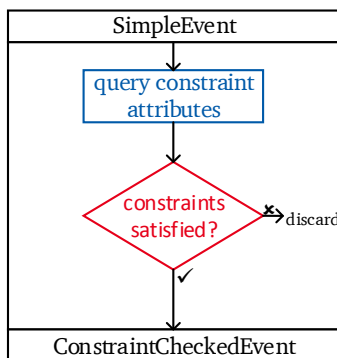


Figure 5.5.: Schematic of a leaf-EEC

```

<PositionEvent>
  <id>5</id>
  <position x="5" y="3"/>
</PositionEvent>

```

```

<GuestPositionEvent>
  <id>5</id>
  <position x="5" y="3"/>
</GuestPositionEvent>

```

Listing 5.4: Example leaf-EEC enrichment

It may happen, that we need the same leaf-EEC for two different leaf-relationship-edges. For example, in the security policy, **person** A has edges to **INSIDE** and **NOT WITHIN**, both using the position attribute (cf. Figure 5.4). In these cases, we generate only one leaf-EEC for both edges. The enriched events

generated by this leaf-EEC are then distributed to all interested relationship nodes, for example by using a publish/subscribe mechanism.

Relationship Nodes

Relationships need to be (re-)evaluated whenever the corresponding attributes change. Since the interaction paradigms for event-based and query attributes are different, we need to generate different EECs depending on the number of edges with an event-based attribute.

One Event-based Attribute: Query-EECs

For relationship nodes which have an event-based attribute and a query attribute (e.g., **INSIDE**) we generate a query-EEC. The query-EEC subscribes to events generated by the corresponding leaf-EEC. Whenever it receives such an event, it queries the external sources for concept instances (e.g., **rooms**) which satisfy the relationship. Furthermore, any such matching instances are checked against the constraints defined in the leaf that is connected via the query attribute (e.g., if the determined rooms' **status='restricted'**). Figure 5.6 illustrates this. While these two queries are conceptually two different steps, we perform them together so we need to query the external sources only once. When a suitable concept instance is found, the event is enriched with the found instance's id (see Listing 5.5 for an example).

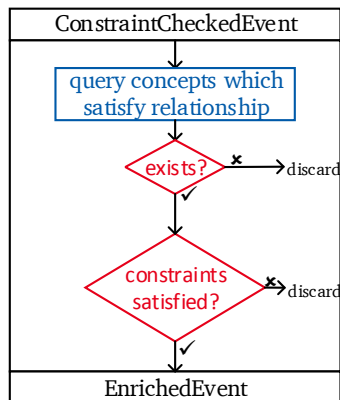


Figure 5.6.: Schematic of a query-EEC

```

<GuestPositionEvent>
  <id>5</id>
  <position x="5" y="3"/>
</GuestPositionEvent>
  
```

```

<AinsideREvent>
  <id>5</id>
  <position x="5" y="3"/>
  <room>R123</room>
</AinsideREvent>
  
```

Listing 5.5: Example query-EEC enrichment

Handling NOT-relationships deserves special attention. Unlike database systems, EBSs cannot use the closed world assumption. This is especially challenging for negative searches (indicated by the NOT operator). Usually, in EBSs, this is handled by injecting specific NOT-events in regular intervals, if the event in question has indeed not been observed.

When evaluating NOT-relationships, we run into a combinatorial problem: assume our example company has 100 employees and 11 restricted rooms and each of these employees is indeed in a restricted room. Thus, every employee is not in 10 restricted rooms and evaluating "employee is not in a restricted room" means $100 \cdot 10 = 1000$ extra NOT-events per interval. It is easy to see that in a slightly larger system this can easily congest the entire system.

We tackle this problem by disabling EECs which deal with NOT-relationships. Only when other EECs detect events, the disabled EECs are activated for a certain time with aliases bound to specific identities. This is best illustrated with our example policy: suppose that the guest Peter (id=5) is detected in the restricted room R123. Upon this detection, the relationship node **B NOT INSIDE R** is activated, with **R** bound to R123. Thus, the query-EEC will process all employee events, but only produce an enriched

event if any of them is inside room R123. Activations are usually time-bound, which mimics a time-based window within which event correlation should happen.

The generated EECs do not send explicit NOT-events. This may seem counter-intuitive at first, so we want to illustrate the reason for this decision: since these EECs are only activated by their parent, said parent can infer the NOT-event when the activation times out. As we will see later when discussing OR-EECs, a single EEC has only a local view and when combining different EECs with logic operators, only their parent can make a meaningful decision if something did not happen.

This approach may cause a problem with events that happen rarely: assume a policy wants to check that certain substances are not too close to each other. However, sensors in the storage only report position *changes*. If substance A is put on a shelf, its position is reported. But substance B's position, which has not changed in the last hour will not be reported within the activation window. The system will consequently not detect a violation, which is incorrect if A and B should not be on the same shelf. For these cases we suggest minimum update intervals which forces event producers to report the last known value every interval (e.g., every second). This overhead is manageable and usually negligible.

When disabling an EEC we have a choice of unsubscribing or dropping events at the EEC side. Unsubscribing has the advantage that the broker has one client less to cater for, while client-side dropping prevents repeated subscription/unsubscription overhead. Our experience shows that activations/deactivations occur frequently and thus decided for client-side dropping.

Two Event-based Attributes: Correlation-EECs

For relationship nodes with two event-based attributes (e.g., **WITHIN**) we generate a correlation-EEC. The correlation-EEC subscribes to events generated by the corresponding leaf-EECs. In contrast to the query-EEC however, it cannot query any missing information. Since both of the relationship function's parameters are event-based attributes, the correlation-EEC has to wait for suitable events to arrive. We therefore keep a time-based window of events for each edge. When an event for one edge is observed, we try finding events from the window of the other edge, for which the relationship evaluation function holds. To evaluate the function, we extract the attribute values from the events. Figure 5.7 illustrates this process. We scan the window starting with the oldest events first and upon a match, delete the contributing event from the window. This is equivalent to a chronicle consumption mode. This consumption mode is the right choice for our middleware as it *"was proposed for applications where there is a correspondence between different types of events and their occurrences, and this correspondence needs to be maintained."* [2]. Correlation-EECs correlate two different kinds of events (e.g., `GuestPositionEvent` and `EmployeePositionEvent`). Furthermore, we want to maintain any detected correlations (which is why we produce a complex event). The recent consumption mode in contrast, is suitable for scenarios *"where events are happening at a fast rate and multiple occurrences of the same event only refine the previous value"* [2]. While we cannot make assumptions about the event rate, we assume that in most policies, new events do not represent mere refinements, but important new data. This is especially true, since events have already been preprocessed by the leaf-EECs.

However, we do not want to rule out the recent consumption mode. To support this mode, policies can be annotated to use a recent consumption mode. In this case, when scanning the window (see above), we do not start with the oldest events, but the newest. Upon finding a match, we then delete the contributing event and all other events with the same identities (cf. Section 5.3.3) from the window. Thus, we have a default consumption mode which should be suitable for most policies, but allow for specifying another mode to provide flexibility.

We do not need to check any constraints at this point, because for event-based attributes, these have already been checked by the leaf-EEC. Upon finding a suitable pair of events, the correlation-EEC produces a new composite event including both original events (refer to Listing 5.6 for an example).

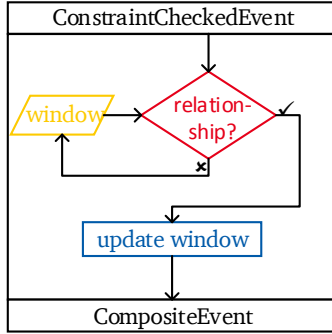


Figure 5.7.: Schematic of a correlation-EEC

`<GuestPositionEvent>` and `<EmployeePositionEvent>`
(details omitted)

```
<AwithinBEvent>
  <GuestPositionEvent>
    <id>5</id>
    <position x="5" y="3"/>
  </GuestPositionEvent>
  <EmployeePositionEvent>
    <id>142</id>
    <position x="4" y="4"/>
  </EmployeePositionEvent>
</AwithinBEvent>
```

Listing 5.6: Example correlation-EEC enrichment

Handling NOT-relationships for correlation-EECs is similar to query-EECs. Disabled by default, they are activated with aliases bound to specific identities. Bound aliases act as a filter for received events. Since either of the two *sides* of the relationship can send an event, this filtering decision may become complex. We created an index which provides a decision in constant time. This is especially important since this decision is on the critical path. We want to illustrate how this index works and why the filtering decision is not trivial in this case.

Assume we have a NOT-relationship with aliases **A** and **B**. For any incoming event, we need to answer the two questions 1) do we need to process the event at all? 2) given an id value of the other side, does the pair satisfy an activation? We illustrate things only for the "A-side" of the relationship; the "B-side" works analogously. Activations are always of the form $A = \alpha; B = \beta$ where A, B denote aliases and α, β values for their ids. Without loss of generality assume that $A = A$. There are two cases which may happen: 1) $B = B$ and 2) $B \neq B$. In the former case, the values for A and B must occur in conjunction, while in the latter case, any value for B is possible, as long as the value for A matches that of the activation. Figure 5.8 illustrates the structure of the index for A. For each activation value α we receive, we create a hashtable entry pointing to a new hashtable. This hashtable contains the corresponding value for β (case 1) above) or a wildcard (case 2) above) with a reference count (right side of the figure). We keep a reference count, because there might be multiple activations for the same value(s) at different times. Since activations time out eventually, we cannot remove the entire entry, as other activations might still keep it active. The hashtable itself is the value of another hashtable with α as the key (left side of the figure). We also keep track of the sum of all reference counts for an id value. In addition to this index, we store a wildcard counter for A and B, which stores the total number of wildcard activations (e.g., the activation $A = 5; B = R123$ is a wildcard activation for B). Table 5.2 illustrates how the index works with an example.

Using the index, we can answer above questions for an event $A = \alpha$ in the following way (again, B follows analogously):

- 1) We need to process the event, iff $A.\text{wildcard} > 0 \vee A[\alpha].\text{sum} > 0$

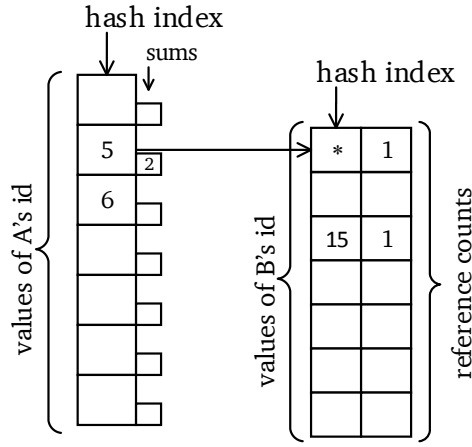


Figure 5.8.: Illustration of the activation index

t	activation	\mathcal{A}		\mathcal{B}	
		index	wildcard counter	index	wildcard counter
t_0	$\mathcal{A} = 5; \mathcal{R} = R123$	$5 \rightarrow \{\star : 1\}$	0	$\star \rightarrow \{5 : 1\}$	1
t_1	$\mathcal{A} = 6; \mathcal{B} = 16$	$5 \rightarrow \{\star : 1\}$ $6 \rightarrow \{16 : 1\}$	0	$\star \rightarrow \{5 : 1\}$ $16 \rightarrow \{6 : 1\}$	1
t_2	$\mathcal{A} = 5; \mathcal{B} = 15$	$5 \rightarrow \{\star : 1, 15 : 1\}$ $6 \rightarrow \{16 : 1\}$	0	$\star \rightarrow \{5 : 1\}$ $16 \rightarrow \{6 : 1\}$ $15 \rightarrow \{5 : 1\}$	1
t_3	$\mathcal{A} = 5; \mathcal{R} = R12$	$5 \rightarrow \{\star : 2, 15 : 1\}$ $6 \rightarrow \{16 : 1\}$	0	$\star \rightarrow \{5 : 2\}$ $16 \rightarrow \{6 : 1\}$ $15 \rightarrow \{5 : 1\}$	2
t_4		$5 \rightarrow \{\star : 1, 15 : 1\}$ $6 \rightarrow \{16 : 1\}$	0	$\star \rightarrow \{5 : 1\}$ $16 \rightarrow \{6 : 1\}$ $15 \rightarrow \{5 : 1\}$	1
t_5		$5 \rightarrow \{\star : 1, 15 : 1\}$ $6 \rightarrow \{16 : 0\}$	0	$\star \rightarrow \{5 : 1\}$ $16 \rightarrow \{6 : 0\}$ $15 \rightarrow \{5 : 1\}$	1

Table 5.2.: Example of the activation index. In this example, activations timeout after 4 time units.

2) Given $\mathcal{B} = \beta$, the activation is satisfied iff $\mathcal{A}[\alpha][\star] > 0 \vee \mathcal{A}[\alpha][\beta] > 0$

Obviously, $\mathcal{A}.\text{wildcard} > 0$ can be checked in constant time. The other checks use one or two hashtable accesses. The cost of accessing an item in a well-designed hashtable is independent of its size and has a constant cost. Thus, either question can be answered in $\mathcal{O}(1)$.

Zero Event-based Attributes: Additional Constraints

Some relationships may not use a single event-based attribute. For example, **room BELONGSTO building** uses the ids of the room and building to determine if a room belongs to a building. Other attributes might be used, but they are most certainly query attributes, because a room's building rarely changes.

As pointed out above, there are no corresponding leaf-EECs for edges with query attributes. Thus, relationship nodes with zero event-based attribute edges cannot be triggered by events. To respect such relationships, we treat these relationship nodes as additional constraints in their sibling query-/correlation-EECs. This is best illustrated with the example policy logic tree in Figure 5.9.

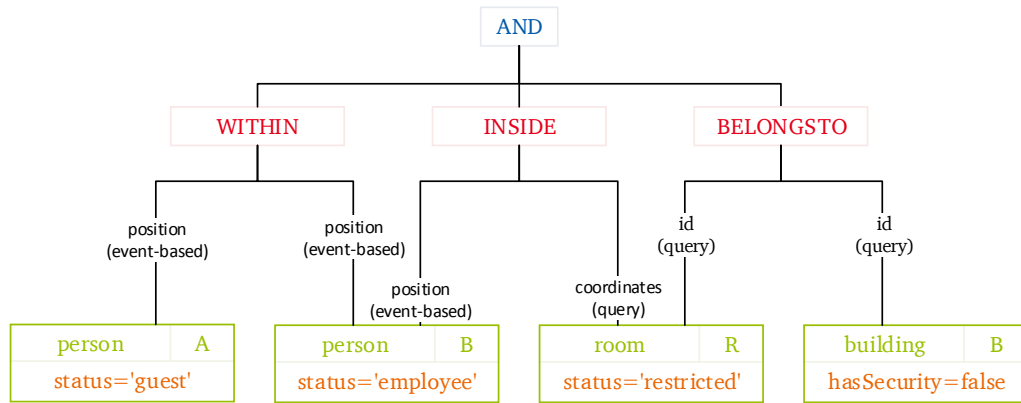


Figure 5.9.: Example of a policy logic tree with a relationship node with zero event-based attributes

In this case, whenever the query-EEC for **INSIDE** processes a room to evaluate if the relationship and constraints hold, it treats **BELONGSTO** as an additional constraint to the room. Thus, even if it finds the room that a person is in and that room has **status='restricted'**, it must also have an associated building which has no security staff. Also, the building is included in the enriched event. The operation of **WITHIN** is not changed, since the corresponding leaves (for aliases **A** and **B**) are not linked to a relationship node with zero event-based attributes.

Followed-by Nodes

EECs for followed-by nodes wait for events representing their left-hand side. Upon detection of such an event, the followed-by-EEC creates a window of time during which it waits for events representing the right-hand side. If such an event arrives within the window, the followed-by-EEC produces a new event containing both the contributing events, otherwise it discards the left-hand side event.

Since child relationship nodes do not send explicit NOT-events as explained in the previous section, NOT-cases are handled in followed-by-EECs in the following way:

$X \rightarrow \neg Y$: Upon detecting X , the EEC sends an activation message to Y (see above at Query-EEC). If the EEC receives a Y -event during the window, it removes the window, since it found a violation

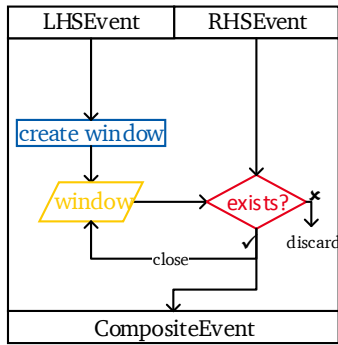


Figure 5.10.: Schematic of a followed-by-EEC

```
<AinsideREvent> → <AlarmEvent>
(details omitted)
```

```
<GfollowedByAEvent>
  <AinsideREvent>
    <id>5</id>
    <position x="5" y="3"/>
    <room>R123</room>
  </AinsideREvent>
  <AlarmEvent>
    <id>1565421</id>
    <room>R123</room>
  </AlarmEvent>
</GfollowedByAEvent>
```

Listing 5.7: Example followed-by-EEC enrichment

to the condition. Only when the window ends without having detected Y, the EEC produces an enriched event.

$\neg X \rightarrow Y$: In this case, the EEC keeps a window of received X-events. As an exception to the rule above (see Query-EEC), the relationship EEC for X is enabled like a normal EEC. Since only positive instances are produced, we do not run into the event-explosion problem outlined above. Upon receiving a Y-event, the EEC checks its window for matching X-events, respecting identity information. If a suitable X-event is found, Y is discarded. Otherwise the EEC produces an enriched event.

$\neg X \rightarrow \neg Y$: In this case, the EEC creates a window whenever one of its children is activated. If it receives an event from one of its children (representing a positive instance), it closes the window. Only when the window times out, the EEC produces an enriched event.

Or Nodes

OR-EECs must distinguish if a received event originates from a NOT-child (e.g., **NOT INSIDE**) or not. Due to the reasons outlined above, we cannot rely on explicit NOT-events. Instead, OR-EECs monitor activations and then try to find a counterexample for any received activation. We believe that we best illustrate this with our running example.

Refer to our example policy logic tree (Figure 5.4) again. Suppose that there is a guest (id=5) in a restricted room (id=R123), which causes the leaf-EEC for **A** to produce an enriched event and consequently the query-EEC for **INSIDE** to do so as well. As we will detail below, this causes the AND-EEC to send an activation message to all EECs with $\{A = '5'; R = 'R123'\}$.

As described above, this message activates the correlation-EEC for **NOT WITHIN** and the query-EEC for **NOT INSIDE**. Thus, the EECs produce events whenever an employee is within 5 m of person 5, or an employee is inside room R123. A first approach may see an employee inside room R123 as a counterexample to **NOT INSIDE** and thus deactivate the EEC. However, this approach fails as Figure 5.11 illustrates.

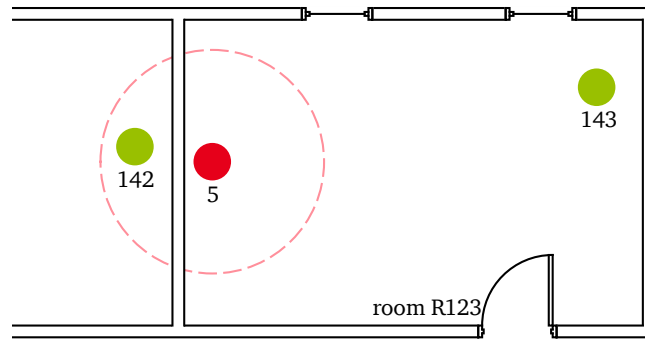


Figure 5.11.: Example why early deactivation does not work

In the depicted setting, an employee with $id=142$ is within 5 meters of the guest and another employee ($id=143$) is inside the same room as the guest. Thus, the query-EEC for **B NOT INSIDE R** produces an enriched event with $\{B = '143'; R = 'R123'\}$. Similarly, the correlation-EEC for **A NOT WITHIN B** produces an enriched event with $\{A = '5'; B = '142'\}$. Both events abide to the bound identities of the activation message. If we deactivated the EECs after they detect these events, we would fail to sound the alarm.

Instead, we need to defer the decision for a counterexample to the OR-EEC. Thus, the relationship-EECs remain activated for the entire activation time. The OR-EEC keeps a window of events for each NOT-child. Upon receiving an event from any of its NOT-children, it attempts to find a match that respects alias identities. If it cannot find such a match, the event is added to the window. If it finds a match, it found a counterexample to *all* its NOT-relationship children while respecting identity information. The contributing events are removed from the window and the corresponding activation is deleted. Once an activation times out, the OR-EEC produces an OR satisfied event, containing the alias identities of the activation. The produced event cannot contain any other events, since it indicates that something did not happen. Table 5.3 further illustrates this.

To summarize this: When receiving events from a "normal" child, OR-EECs immediately produce an OR satisfied event, including the triggering event. When receiving events from NOT-children, OR-EECs try to construct a counterexample for their activations. Only when they fail to construct one, in the time given by the activation, they produce an OR satisfied event. Relationship-EECs cannot construct counterexamples themselves, since they only have a local view.

And Nodes

As already said above, there is only one AND node per policy. The generated AND-EEC waits for events from its immediate children. Upon detection of such an event, the AND-EEC sends an activation message to all other EECs. It then waits for a certain time for events from the other children to arrive. If all children have sent events with matching identities within this time frame, it produces an event that will trigger the policy action(s). Technically this waiting is realized by putting received events into a separate cache per child. Upon receiving an event, the EEC tries to match it with events from the other caches while respecting identities. Matches are found by using a natural join-like procedure, which is illustrated in Table 5.4. To keep intermediate results as small as possible, we first match those sets of events which share aliases. For example, when receiving an event for child3, it is first matched against all events from child2, before proceeding to child1.

t^{\S}	messages	activations	NOT inside	NOT within	operation
...			$\{B = '101'; R = 'R12'\}_{-4}$		
t_0	activation: $\{A = '1'; R = 'R12'\}$	$\{A = '1'; R = 'R12'\}_0$	$\{B = '101'; R = 'R12'\}_{-4}$		
t_1	BinsideREvent: $\{B = '101'; R = 'R12'\}$	$\{A = '1'; R = 'R12'\}_0$	$\{B = '101'; R = 'R12'\}_1$		*
t_2	activation: $\{A = '5'; R = 'R123'\}$	$\{A = '1'; R = 'R12'\}_0$ $\{A = '5'; R = 'R123'\}_2$	$\{B = '101'; R = 'R12'\}_1$		
t_3	AwithinBEvent: $\{A = '1'; B = '105'\}$	$\{A = '1'; R = 'R12'\}_0$ $\{A = '5'; R = 'R123'\}_2$	$\{B = '101'; R = 'R12'\}_1$	$\{A = '1'; B = '105'\}_3$	
t_4	AwithinBEvent: $\{A = '1'; B = '101'\}$	$\{A = '5'; R = 'R123'\}_2$		$\{A = '1'; B = '105'\}_3$	†
t_5	AwithinBEvent: $\{A = '5'; B = '142'\}$	$\{A = '5'; R = 'R123'\}_2$		$\{A = '1'; B = '105'\}_3$ $\{A = '5'; B = '142'\}_5$	
t_6	BinsideREvent: $\{B = '143'; R = 'R123'\}$	$\{A = '5'; R = 'R123'\}_2$	$\{B = '143'; R = 'R123'\}_6$	$\{A = '1'; B = '105'\}_3$ $\{A = '5'; B = '142'\}_5$	
t_7			$\{B = '143'; R = 'R123'\}_6$	$\{A = '1'; B = '105'\}_3$ $\{A = '5'; B = '142'\}_5$	‡

[§] We assume a window size of 5.

* The NOT inside event moved out of the window.

† Match found: $\{A = '1'; B = '101'; R = 'R12'\}$ which leads to the removal of the activation $\{A = '1'; R = 'R12'\}_0$.

‡ The activation $\{A = '5'; R = 'R123'\}_2$ timed out: produce an OR satisfied event.

Table 5.3.: Illustration of activations and NOT-relationships interplay

child1	child2	child3	matches
$\{A = '2'; B = '7'\}$ $\{A = '3'; B = '5'\}$	$\{B = '5'; C = '13'\}$	$\{C = '15'; D = '8'\}$ $\{C = '13'; D = '7'\}$	$\{A = '3'; B = '5'; C = '13'; D = '7'\}$
$\{A = '2'; B = '7'\}$	$\{B = '5'; C = '15'\}$	$\{C = '15'; D = '8'\}$	no matches
$\{A = '2'; B = '7'\}$	$\{B = '7'; C = '15'\}$	$\{C = '10'; D = '8'\}$	no matches
$\{A = '2'; B = '7'\}$ $\{A = '3'; B = '5'\}$	$\{B = '5'; C = '15'\}$ $\{B = '5'; C = '15'\}$	$\{C = '15'; D = '8'\}$	$\{A = '2'; B = '7'; C = '15'; D = '8'\}$ $\{A = '3'; B = '5'; C = '15'; D = '8'\}$

Table 5.4.: Illustration of the match-finding algorithm. The recently received event is in **bold**.

5.3.4 Settling: Policy Shutdown

Upon receiving a policy shutdown request, the middleware sends a shutdown event to the command channel of the policy. Each EEC of this policy receives this event and unsubscribes from any events and then shuts down.

5 Architecture

CPSs typically rely on publish/subscribe MOM to disseminate events. Thus, we build the middleware for our approach on top of a MOM. Figure 5.12 provides an architectural overview of the components for our approach. We do not describe every component in detail here, since their functionality has been described in the previous sections and the focus here is on their interaction.

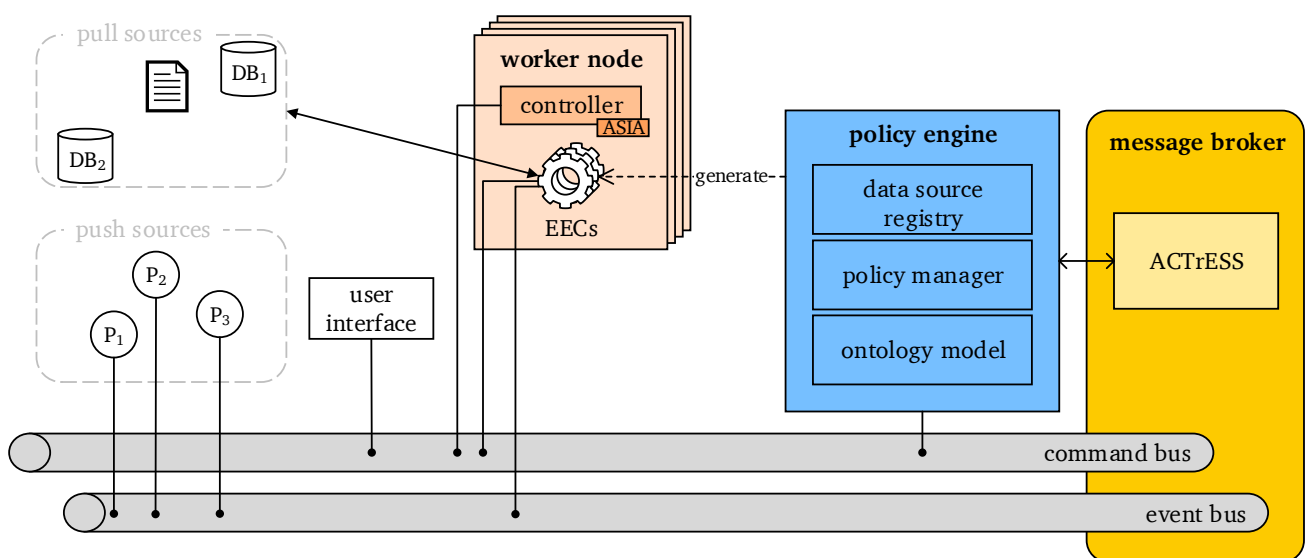


Figure 5.12.: Architecture for the DPL-enabled middleware

5.4.1 Event and Command Bus

We use the messaging functionality of the MOM to let components communicate in a decoupled and distributed way. We distinguish between a command bus and an event bus. The command bus is used to coordinate components like sending a new policy to the middleware. The event bus distributes the primitive and enriched/composed events.

5.4.2 User Interface

The user interface allows users to interact with the system. It allows for creating, altering and deleting policies.

When a user wants to create a new policy, they simply write it and send it to the middleware. To assist the user with direct feedback, we built an Eclipse plugin with content assist, syntax highlighting and automatic syntax checking. Figure 5.13 shows a screenshot of the policy editor and the ontology model editor. Please note that these editors are usually used by different users.

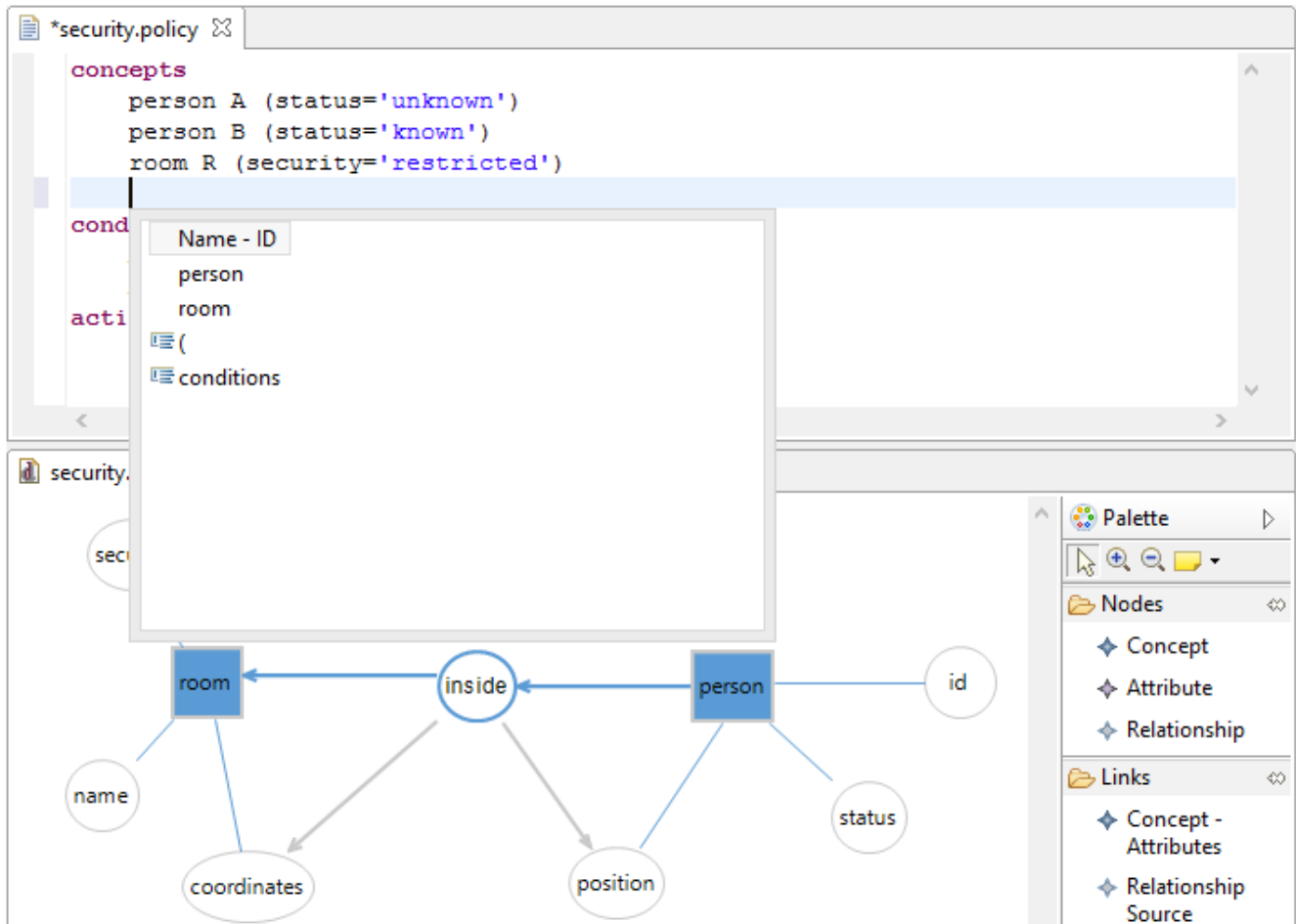


Figure 5.13.: Screenshot of the policy editor and ontology model editor.
The policy editor shows an example of content assist.

This enables the user to quickly see which concepts and relationship functions are available and prevents them from sending erroneous policies to the middleware. One can even imagine having a graphical policy editor, similar to graphical SQL query editors (e.g., Microsoft Access). No matter which tool is used, after creating the policy, the user assigns a unique name to it and uploads it to the middleware using the command bus, which will then take care of processing.

There are different reasons why a user might want to alter an existing policy. It might simply be the case that the user wants a different behavior or because the underlying ontology changed. In the latter case, some policies might have become invalid. These policies are then displayed to the user for review, with a short explanation why they have become invalid. In any case, the user can retrieve a policy by its name, make the necessary adjustments and send it back to the middleware.

At some point, a policy might no longer be needed. Similar to altering a policy by referencing it with its name, a user can delete a policy from the middleware. We refer to this as an explicit deletion. Processing of the policy then simply stops. Users might also want to delete policies automatically when certain conditions are met. In this case, they simply need to issue the delete command in the actions part of any policy. We then speak of implicit deletion. It might be sensible to implement some security/authentication in multi-user environments to protect policies against unauthorized manipulation. While certainly possible (e.g., storing access rights with policies), such measures are not the focus of this thesis.

5.4.3 Policy Engine

The policy engine is the heart of our middleware. It serves as a coordinating component in a controller-like fashion: Upon receiving a policy, the policy engine uses the policy manager to analyze the policy. Based on the analysis, it generates EECs (see Section 5.3.3 for details).

Data Source Registry

The data source registry keeps track of all information sources. It uses the metadata provided upon registration of data sources to provide unified access to pull sources. Push sources use the event bus to deliver information. Both kinds of sources can be added at runtime to the data source registry.

Policy Manager

The policy manager keeps track of all registered policies. It is responsible for handling new and edited policies, as well as ensuring proper shutdown of all related components when a policy is deleted. Policy ownership by specific users along with access rights, as discussed in Section 5.4.2 can be implemented here.

Furthermore, the policy manager keeps references to all generated EECs. Thus, upon policy shutdown, the policy manager knows which EECs to shut down.

Ontology Model

The ontology model component provides access to the structure of the ontology model. As described above, this information is used during creation of the policy logic tree and during code generation. Furthermore, it is responsible for providing the relationship evaluation functions (cf., Definition 8).

We allow for modifications to the ontology model at runtime. However, this is a costly operation as policy logic tree and the code have to be generated again. In addition, changes to the underlying ontology may break existing policies. Yet we do not want to exclude this option.

5.4.4 ACTrESS

We talked about ACTrESS extensively in chapter 4. For this middleware, we make use of ACTrESS. As pointed out in Section 5.3.1 we assume events to be in the format specified by the ontology and use ACTrESS for transforming any deviating events.

We want to illustrate this with an example: our example company has deployed several position-tracking sensors in their facilities. Among others, RFID-readers and cameras report position information of persons. However, both sensors report positions relative to their own location, the room coordinates in the database are in absolute (company-wide), Cartesian coordinates. Position information from the sensors has to be translated to absolute coordinates. In addition, the RFID-readers report positions in Cartesian $(x; y)$ coordinates, while the camera provides them in polar coordinates $(r; \theta)$. Obviously, these two formats cannot be treated uniformly. The interpretation context of the RFID-reader will say that the format is `relative.cartesian`, while the camera's interpretation context will say `relative.polar`. The generated events are handed to ACTrESS which outputs absolute Cartesian coordinates. Thus, no matter the origin, the policy engine can assume homogeneous units and structure of events.

5.4.5 Worker Nodes

We designed EECs so they do not need to run on the same machine as the middleware. Each worker node has a controller that is able to receive new EEC definitions and execute them. There can be an arbitrary number of EEC running on a worker node. This allows for a high amount of parallelization and enables easy load balancing. This is especially useful for modern cloud environments, in which more processing units can be spawned on demand (be it through scale-up or scale-out). The beauty of this approach is, that it also allows for easily scaling down and in as well: when the utilization of multiple processing units is below a certain threshold, their work is assigned to just one node and the remaining nodes are released, thereby saving money for the cloud tenant. Of course such an approach leads to more messages that the broker has to distribute, compared to a monolithic approach where all EECs are executed on a single, powerful machine, exchanging enriched events directly via method calls. To mitigate this, distributed brokers or direct (socket-based) connections between communicating EECs can be used.

Distribution of computation forces considerations about the impact of distributed clocks. We therefore make the following assumptions: 1) all worker nodes run inside the same data center; 2) worker nodes have access to a Network Time Protocol (NTP) server. We do not see reasons to distribute work across data centers, where transmission delay between individual nodes is significantly larger than within one data center. Even in cloud environments, tenants are able to specify in which data center their virtual servers should be spawned. Furthermore, almost all machines today have Internet access and can talk to an NTP server. Even if no direct Internet access is possible, it is safe to assume that a local NTP server exists. NTP-synchronized clocks deviate only by microseconds [139]. This allows for ordering events according to the model of Liebig et al. [128]. Due to the bounded and low clock deviations, uncertainty in event order is unlikely to happen. Event order is only important for followed-by nodes. In case of uncertainty we always decide against the followed-by condition. We believe that in case of followed-by, users want to have absolute certainty about the event relationship. If this certainty is not required they can use conjunction instead.

Approximate Monitoring: ASIA

To achieve dynamic load balancing, we need to collect monitoring information about the worker nodes. We collaborated with other researchers to create a system for scalable dissemination of monitoring information in distributed systems [93]. The system called ASIA, allows subscribers to specify a certain imprecision they are willing to tolerate in order to reduce the total number of messages. The imprecision can be changed at runtime. This allows for aggregating messages.

For example, each worker node produces monitoring events about the current CPU utilization. The policy engine responsible for load balancing is only interested in the average CPU utilization across the last 60 seconds and is willing to tolerate an imprecision of 30. This means that a monitoring event reporting that the average CPU utilization is at 50 %, means that it can in fact be anywhere between 20 % and 80 %. In addition, ASIA can aggregate the single CPU utilization readings to an average value.

Upon reaching a certain threshold (e.g., 60 %), the policy engine can tell ASIA to decrease the imprecision for the worker node whose CPU utilization exceeded the threshold, to get more precise data. If CPU utilization is indeed too high, the policy engine can instantiate more worker nodes or shift the load and then increase the imprecision again.

5 Implementation

We implemented our approach using Java Message Service (JMS) as the messaging API. We decided for ActiveMQ as the messaging middleware, but as our architecture diagram (cf. Figure 5.12) shows, we do not depend on a concrete implementation.

Tools

We used Xtext¹ to generate our policy parser and Graphical Modeling Framework (GMF)² to build the ontology editor. Both rely on Eclipse Modeling Framework (EMF)³, which provides a metamodel (called Ecore) to specify domain models. These can then be used to generate Java classes. EMF brings built-in support for (de-)serialization of models using XML Metadata Interchange (XMI) [147]. Thus, users do not have to rely on our ontology editor and can instead use their own, XMI-compatible one.

GMF is a framework enabling the creation of custom graphical editors. We used this framework to build our ontology editor (see the lower half of Figure 5.13). Ontologies created with the editor are serialized using XMI and can be stored in disk or sent to the policy engine. Loading an XMI serialization automatically instantiates appropriate classes (e.g., **Concept**, **Attribute** and **Relationship**).

Xtext is a popular parser generator for domain specific languages. We translated our grammar to an Xtext-compatible one and generated a DPL parser from that. When processing a policy, the parser instantiates appropriate classes (e.g., **Alias**, **Condition**, etc.).

Window Sizes

During the introduction to the various EECs, we mentioned windows multiple times. We have not yet talked about their size. A sensible window size always depends on the concrete application. In the current implementation, the computer expert setting up the system can configure the window size. For a productive system, this is too static of course, but our approach remains the same when switching to more dynamic window sizes. These might be explicitly stated or inferred from the policy (see Section 7.1).

Instantiation Sequence

Figure 5.14 illustrates the information flow between users, the policy engine and the worker nodes. As mentioned above, an expert designs the ontology model and sends it to the policy engine, including the model classes themselves. Model classes are those that the ontology model refers to (e.g., **Positions** and **Coordinates**). The policy engine stores the ontology and loads the model classes with our custom classloader which supports loading class bytecode from main memory.

When a domain expert sends a policy, the engine parses it and sends the policy logic tree (see Section 5.3.2) to the generator. The generator generates EEC code as described in Section 5.3.3. The generated code is then passed to the compiler which compiles it to runnable bytecode. We had to extend various Java core classes which are used for runtime compiling, because out of the box, these classes only support loading and compiling classes from and to disk. However, the model classes that the generated code depends on (and are thus needed for compilation) only exist in main memory. Likewise, since we want to distribute the bytecode later, we wanted to avoid the compilation to disk and then reading from

¹ <https://eclipse.org/Xtext/>

² <https://www.eclipse.org/gmf-tooling/>

³ <https://www.eclipse.org/modeling/emf/>

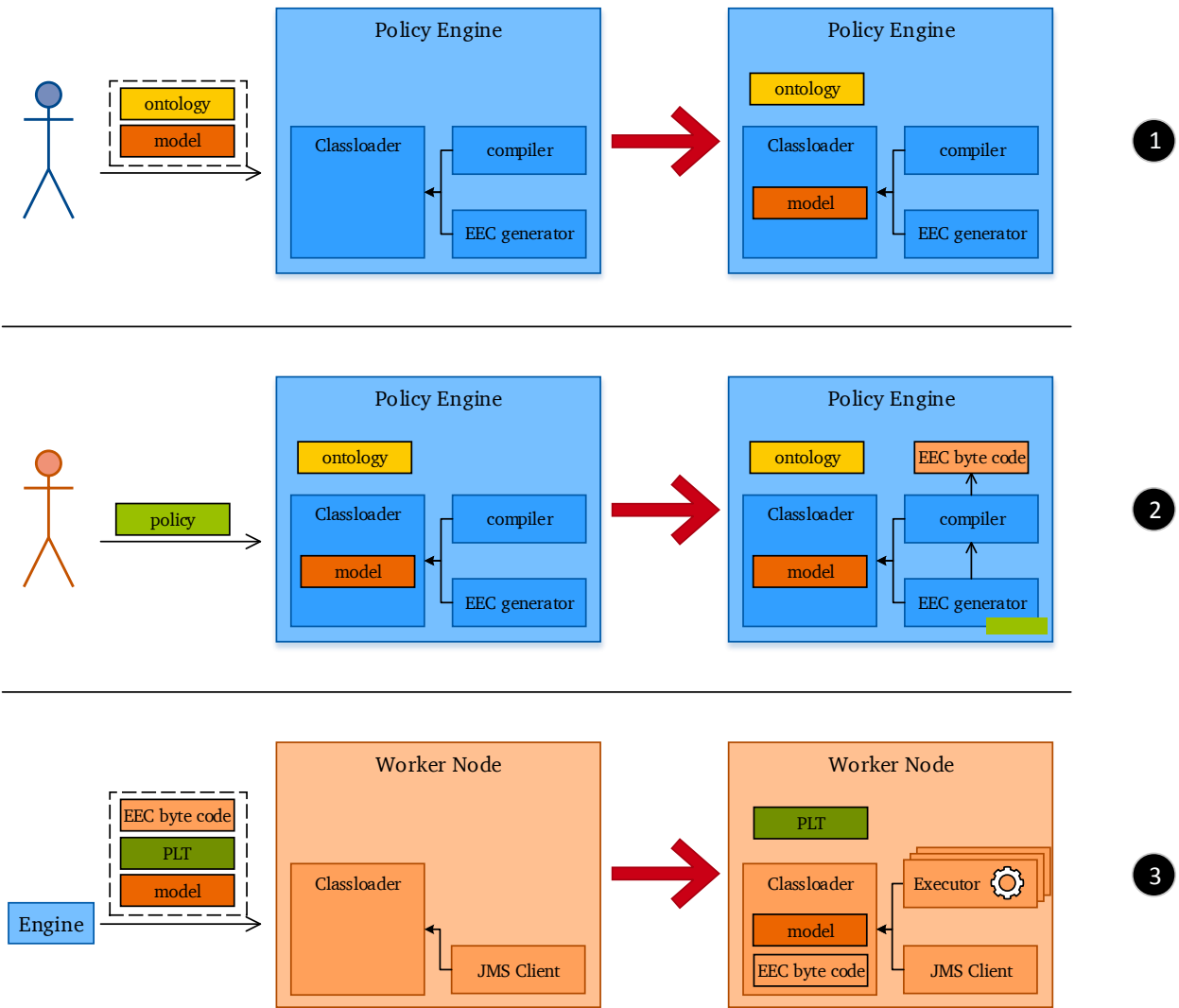


Figure 5.14.: Sequence of the contextualization process starting with the registration of the ontology.

disk again. Our extensions are not project specific and can easily be used by any other Java project. In fact, ACTrESS uses the same classes for its generation and compilation steps.

After generating EEC bytecode, the policy engine sends the policy logic tree, the model classes and the EEC bytecode to the worker nodes, along with commands indicating which worker node is responsible for which tree node. Sending all code to all nodes may seem excessive. However, it allows for easier load balancing later, as worker nodes can instantiate more EECs without needing any more code. Of course, in environments where memory is a premium, this can easily be changed so worker nodes only get the bytecode they are supposed to run. The worker node uses our custom classloader to load the model and EEC bytecode from main memory and then instantiates as many EECs as it was told to. Note that we can run EECs of multiple policies on a worker node. In this case, we instantiate a classloader per policy to ensure isolation.

So far, we relied on the broker's messaging facility to send messages between EECs. While this is certainly possible, using a publish/subscribe system for communication between known participants is ill-suited. In joint work with Prashanth Sadanand, we implemented socket-based communication between EECs [174].

As outlined above, each worker node receives the EEC bytecode and the policy logic tree. In addition, worker nodes are also informed about which other worker nodes run which EECs. Each node can then run and configure its EECs with the addresses of the EECs higher up the tree. After this bootstrapping process has finished, operation resumes as before. EECs use a socket-based communication whenever they have an active socket connection, and use JMS only as a fallback.

Please note that our socket-based communication does not replace JMS entirely. We still use JMS to distribute commands and for informing the leaf-EECs about new events. As our evaluation shows (see Section 5.6.4) using sockets increases throughput by about 15%. The Master Thesis of Mr. Sadanand provides more details [174].

5 Evaluation

We evaluated the usability and performance of our approach. The usability evaluation includes a theoretical evaluation using the established Cognitive Dimensions framework, and a user study we conducted. In the usability evaluation, we compare our approach –referred to as DPL– to Esper and Java. The performance evaluation compares the performance of our approach to that of Esper.

5.6.1 Scenarios

We use scenarios to evaluate the various aspects of our approach. We will briefly introduce them so readers have something to relate to in the following sections.

Our first scenario follows our running example of security concerns. The policy in DPL used for the evaluation is given in Listing 5.8. We provide the stream queries in Event Processing Language (EPL)

```
concepts
  person A (status='guest')
  person B (status='employee')
  room R (security='restricted')
conditions
  A is inside R
  A is not within 5m of B or B is not inside R
actions
  raise alarm
```

Listing 5.8: Security example in DPL

that realize the same functionality using Esper in Listing 5.9. We believe most readers to be familiar with Java so we do not provide the Java code here and refer to the appendix.

The second scenario is from the domain of logistics. Transported goods need to be monitored for various conditions, e.g., temperature. There are, however, more complicated constraints than temperature

```

insert into UserStream
  select * from pattern[timer:interval(0)],
  sql:db1 ['select * from rfid_tags']

insert into RoomStream
  select * from pattern[timer:interval(0)],
  sql:db1 ['select * from room']

insert into EmployeeEvent
  select p.id, p.position
  from PersonPositions as p, UserStream as u
  where u.id = p.id
        and u.status = 'employee'

insert into GuestEvent
  select p.id, p.position
  from PersonPositions as p, UserStream as u
  where u.id = p.id
        and u.status = 'guest'

insert into EmployeeRoom
  select e.id, e.position, r.id as room
  from EmployeeEvent as e, RoomStream as r
  where Functions.inside(e.position, r.coordinates)

insert into GuestRoom
  select g.id, g.position, r.id as room
  from GuestEvent as g, RoomStream as r
  where Functions.inside(g.position, r.coordinates)
        and r.security = 'restricted'

insert into AlarmEvent
  select a.room, a.id, a.position from pattern
  [every a=GuestRoom -> (timer:interval(5 sec) and not b=EmployeeEvent
    (Functions.within(position, a.position, 5.0) and room = a.room)
  )]

```

Listing 5.9: Security example in EPL

thresholds. One such constraint concerns potentially dangerous liquids. Because some liquids create powerful reactions when combined, they must not be put too close to each other. In this scenario, a logistics provider wants to ensure this requirement, by checking any two dangerous substances on the same truck for their proximity. If a violation is detected, headquarters should be informed immediately to decide upon further action. For this scenario we assume that trucks are outfitted with location tracking capabilities to know the locations of their freight items. We list the pieces of code for this scenario in the appendix.

5.6.2 Cognitive Dimensions

Greene and Petre established a framework called *Cognitive Dimensions* which serves as an evaluation technique for notations and programming languages [102]. The framework defines thirteen dimensions along which a notation can be judged. Since development environments usually have a significant impact on the cognitive load of the programmer, some of these dimensions can only be sensibly used for the notation in conjunction with its development environment. Greene and Petre acknowledge that a notation cannot shine in all of the dimensions as often tradeoffs between the different dimensions have to be made.

We analyzed our language DPL, Esper's EPL and Java along these dimensions. For DPL, we assumed the editor we created (see Figure 5.13). EPL does not have editor support, but we were able to find a syntax highlighter for Eclipse, so we assume the users use this, instead of just a text editor. For Java we assume a powerful IDE like Eclipse. Please note that the following analysis is a discussion of the languages' features along the introduced dimensions. As it is often the case when evaluating usability, in such an analysis it is impossible to present hard, measurable facts. However, we hope that readers find our arguments and conclusions comprehensible and compelling.

Abstraction Gradient

Abstraction gradient refers to the minimum and maximum levels of abstraction a language allows or enforces. The authors provide the classifying categories *abstraction-hating*, *abstraction-tolerant* and *abstraction-hungry*.

DPL does not allow for new abstractions to be introduced. The end user has to work with the concepts provided by the ontology model. Other abstractions are also fixed, for example the structure of conditions. Thus, DPL is abstraction-hating. In contrast to that, EPL forces the user to create abstractions. The user has to introduce new streams as abstractions of existing ones, building more and more complex representations, serving as higher abstractions. Thus, EPL is abstraction-hungry. Like most modern high-level programming languages, Java is abstraction-hungry as well, because it requires the abstraction of objects, which form inheritance hierarchies.

We should point out that abstraction-hating is not better or worse than abstraction-hungry, as each of these provide their own unique benefits. Greene and Petre point out that "many potential end-users are repelled by abstraction-hungry systems", while also stating that abstractions help with *viscosity* and reducing *error-proneness*, which are two other cognitive dimensions. Since our focus is on end-users, especially those that are not used to creating abstractions, we think that for our goals, an abstraction-hating language is the right choice.

Closeness of Mapping

When using notation, one always tries to map a real world problem to the notation's model. This dimension states that this is easiest if the problem can be mapped directly to the notation. Thus, the closer the notation model is to the user's mental model of the problem or task, the easier it is to use a notation/language.

Java, like all major programming languages, has myriad of language elements that do not map to the problem at all: curly braces, keywords like `public` or `void`, etc. require extensive language training just for writing syntactically correct programs. EPL uses streams as its main abstraction. The problem is that humans do not think in terms of streams when picturing how real world elements interact. Still, streams provide a tangible abstraction, albeit not a natural one. DPL has a comparatively close mapping. We looked into psychological literature on cognition and found that humans classify objects in the world into categories with defining attributes and relate categories to each other by using these defining attributes [165]. For example, a chair is something to sit on that is often found near a table. Thus, formulating policies along concepts with their attributes and their relationships to each other seems to be a close mapping to how humans actually think.

Consistency

Since consistency is hard to define in a crisp and manageable way, Greene and Petre adopt the working definition that consistency is a measure of how well parts of the language can be inferred if the user already knows some language constructs.

Java is somewhat consistent, the structure of methods, loops and conditionals are similar to one another. However, constructions like annotations show that the language has evolved a long way and new concepts cannot be easily integrated into the core part of the language, forcing new, inconsistent language elements. EPL is surprisingly consistent. Stream declarations are consistent and have the same, reoccurring structure. Window definitions can appear at various places, but always with the same syntax. It should be possible to declare a stream just by seeing how another stream is declared. Only advanced concepts like patterns cannot be guessed. With above definition, DPL is an inconsistent language. The reason is that the syntax itself is very small and thus does not have any parts that could be inferred. The major parts of the syntax (e.g., concept definitions, conditions, actions) refer to entirely different things. On the plus side, within each part, declarations are very consistent.

Diffuseness / Terseness

Languages that are terse can express a certain meaning with fewer symbols than diffuse languages. Of course, languages with a closer mapping to the problem domain can be more terse, as the language constructs may be readily available. However, like Greene and Petre we try to evaluate terseness independently of closeness of mapping and use their approximation to count the number of entities. Again, it is not entirely clear what an entity is, other than a significant element in the code.

Java is not terse at all. Usually, multiple files have to be inspected to get an understanding of what is going on. In a way, Java pays with diffuseness for its generality. At least, most methods and classes in Java have visually different appearances, if the programmer follows the indentation guidelines. If we assume the entities type, variable, assignment, annotation, method declaration, method parameter, method call, condition, logical connector, type cast and loop initialization, the necessary code for the example policy has 217 entities⁴.

⁴ We present the code in the Appendix

EPL is terser than Java. Stream queries usually fit into one file, so only one file needs to be scanned. However, often queries are distributed across the various files of the host language and thus, it becomes very difficult to find those parts, especially since appropriate search support is missing from development environments. Since streams often depend on each other, this can quickly become a burden for the user. For entities we counted 'insert into', stream identifier, 'select', attribute, 'from table', 'pattern', 'timer', 'interval', where-condition-attribute, where-condition-value, logical connector, function, function argument, 'every', pattern-assignment, 'not', '->'. This results in 91 entities for the EPL equivalent stream queries of our example policy.

DPL is very terse, policies fit into a few lines and thus easily on a single screen page. In addition, it has clear sections (concepts, conditions, actions) so it is easy to identify the individual parts. It is, however, not overly terse, which would be a problem, as all policies would visually look the same and scanning them would become impossible. Figure 5.15 illustrates this with two example policies. The main entities we identified are concept+alias, attribute, attribute value, relationship function, function argument, logical connector, 'not' and action. This results in 23 entities for our example policy.

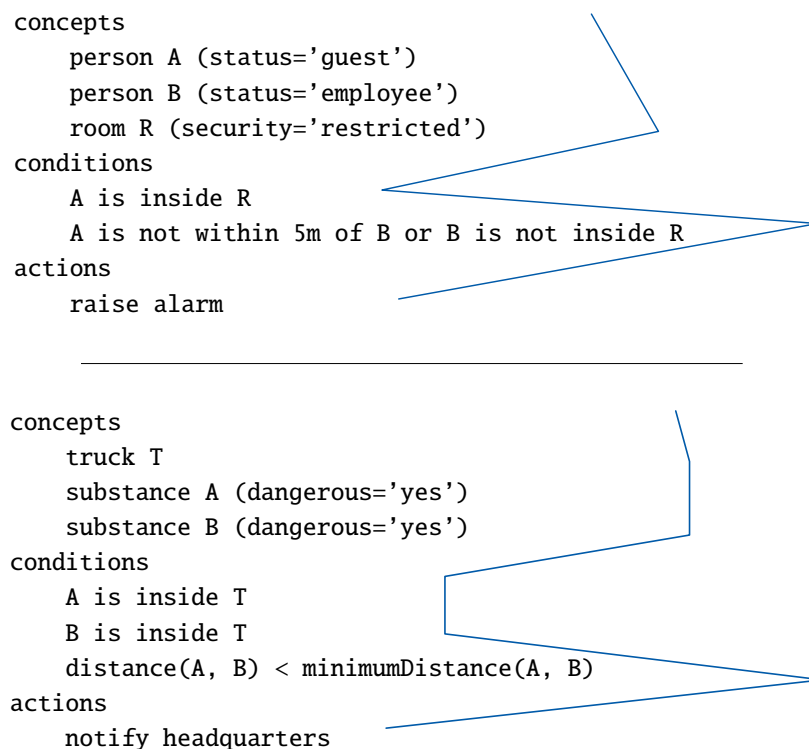


Figure 5.15.: DPL is not overly terse. Showing two example policies together with their visual scan lines.

Error-proneness

This cognitive dimension focuses on slips, rather than conceptual errors. Slips are those kind of errors that happen, although the user knows what to do, e.g., mistypings.

All three languages suffer from error-proneness like most text-based languages do. Mistypings can happen in Java, EPL and DPL, although this is alleviated by syntax highlighting and auto-completion. However, when identifiers are similar, mistypings are probably not noticed and can cause hard-to-trace errors. Likewise, all languages suffer from the "paired delimiter" problem: parenthesis, quotation marks

and other delimiters have to appear pair-wise which can lead to unintended behavior. Again, automatic insertion by the editor helps, but does not eliminate this problem. Java and EPL are a little worse on this dimensions, because they have many more kinds of paired delimiters, increasing the overall complexity.

Hard Mental Operations

Greene and Petre pose two conditions for something to count as a hard mental operation: (1) the mental operation must be difficult due to the notation and not the expressed meaning and (2) combining two or three of such operations drastically increases difficulty. As an example for (1) they mention conditionals and negatives, arguing that research has shown that decision tables are much easier to handle and thus the problem lies within notation. For (2) they mention multiple negatives and complex booleans.

Again, all three languages expose the same problem: complex boolean expressions. In the case of DPL, we believe that the conditions will not become overly complex. Still, it may be worthwhile to explore the benefit of decision tables as part of the policy specification. However, we see this as out of scope of the thesis and see it as a subject of future work. EPL introduces another source for hard mental operations: patterns. Depending on where elements like 'every' and 'timer:interval' are within the pattern, the meaning of what is detected changes significantly. Even the Esper documentation has to spend a lot of pages to explain the different meanings⁵.

Hidden Dependencies

Hidden dependencies occur when a component depends on another one, but this dependency is not entirely clear. Greene and Petre provide spreadsheets as an example. While it is usually very clear from which cells a cell's formula takes its values, there is no obvious way in seeing which other cells depend on the current one. Precisely for this reason, Microsoft introduced formula spies to Excel, which allow for seeing these dependencies, making them no longer hidden.

The same hidden dependencies as in spreadsheets occur in EPL. It is easy to see which streams a certain query depends on, but it is very hard to determine which other queries depend on a given stream definition. Thus, in order to reveal such dependencies, all stream queries have to be analyzed. Since they may be distributed across several files, whose existence may even be unknown, this becomes a hard problem. Java exhibits similar hidden dependencies, where other classes may depend on a given class. However, most modern programming environment provide reference searchers, similar to Excel's formula spies. Since policies are self-contained, DPL does not suffer from hidden dependencies. While conditions depend on concept/alias definitions, this dependency is very clear and usually just a few lines apart. Even if we reuse enriched events between policies, this dependency is middleware-managed and thus not exposed to the user.

Premature Commitment

This dimension refers to how many decisions have to be made, before their effect becomes clear. Greene and Petre illustrate this with the example of trying to write a document index before actually writing the document. They provide two indicators to look for premature commitment: dependencies among components and an enforced order in component creation.

⁵ http://www.espertech.com/esper/release-5.2.0/esper-reference/html/event_patterns.html#pattern-operators

Java suffers from premature commitment a bit. Programmers have to declare a method, including its signature, before they can use it. Likewise, any classes must be declared before they can be used. This does not pose a huge problem in practice, because powerful refactoring tools exist. However, this is a matter of viscosity (see below) and not premature commitment. EPL and DPL also force a certain amount of premature commitment. In EPL, stream identifiers and their structure must be defined before being used. Although first declaring an empty, dependent stream to see what information it needs is possible it is not executable. DPL requires the declaration of concepts/aliases before they are used in the conditions. While making adjustments is very easy, this remains a matter of viscosity and strictly speaking DPL suffers from premature commitment. A possible solution could be to modify the editor so that while writing the conditions, any non-declared aliases are automatically inserted.

Progressive Evaluation

Progressive evaluation refers to the ability to execute partial programs.

Strictly speaking, Java does not support progressive evaluation. However, most development environments help by automatically creating method stubs, and thereby generating compilable code which can be partially evaluated. EPL supports progressive evaluation almost naturally. The output of individual streams can always be inspected by adapting the listener in the host language without any need to define more complex streams. In addition, complex stream definitions can be fed with dummy data for debugging purposes. DPL on the other hand, provides little to no support for progressive evaluation. It is of course possible to submit a policy that misses some conditions and see what happens. We even expect novices to do so in order to get familiar with how policies work. But there is no support beyond that. This lack of support for progressive evaluation, however, is by design. We claim that users think of a situation as an atomic chunk, which they do not even want to evaluate in parts. Since policies are essentially situation descriptions, they serve as these atomic chunks.

Role-Expressiveness

Role-expressiveness is a measure how easy it is to identify what a piece of code does. Research shows that role-expressiveness is increased by meaningful identities and the presence of distinct "beacons". Beacons are code structures that are highly characteristic.

DPL has those beacons: the keywords "concepts", "conditions" and "actions" serve as beacons, clearly identifying the roles of the following statements. In addition, DPL is comparatively prosaic and thus expresses the meaning of elements within the language e.g., "A is inside R" does not need any further explanation. In EPL, stream definitions express their role well. We see the benefit here in the declarative nature of the language. Problematic again are pattern definitions, where the roles of elements (e.g., 'every' and 'timer:interval') are obscure. Java does worst in role-expressiveness. Almost nothing in the language suggests its function. Classes, methods and variables are all concepts which have to be learned before it is clear what they do. Exceptions are language constructs for branching (if) and loops (for/while), which suggest their meaning. However, on the worst end are operators like '?', which leave even experienced programmers puzzled at first⁶.

⁶ Of course this does not mean that the operator is not useful, in fact the author finds that it allows for creating elegant and readable expressions

Secondary Notation and Escape from Formalism

This dimension is a notion for any mechanism that allows for expressing the programmer's intent, but is not part of the formal syntax. For textual languages, two famous kinds exist: whitespace and comments. Whitespace can be horizontal (i.e. indentation) or vertical (i.e. grouping semantic units in paragraphs).

Java, EPL and DPL support whitespace as well as comments as a form of secondary notation. For Java there even exist world wide conventions for indentation and entire API documentations are rendered from specially formatted comments.

Viscosity

Viscosity is originally a term from physics/chemistry to describe a fluid's resistance to deformation. In the Cognitive Dimensions framework, viscosity describes how much work it is to make a small change, for example rename a component. According to Greene and Petre, a standard example is having to make a global change by hand due to the lack of a global update tool. Because it is standard business in programming to change and evolve a program, it is important for languages to be *fluid*.

Java with a standard text editor would be a highly viscous language. However, with today's programming environments providing powerful refactoring tools, which are able to extract interfaces from classes in an automated fashion, Java is in fact fluid. In contrast, EPL suffers from viscosity. For example, changing a stream's name requires manual replacement at every place it is used. If all stream definitions are in one file, this is just a bit cumbersome. However, if definitions are distributed across files, this requires identification of those files first and then a manual update. To be fair, we see this not as an inherent problem to the language itself, but to the lacking support of environments. On the other hand, DPL does not have elaborate editor support, but the provided editor supports basic renaming. Since policies are atomic, there are no global changes and thus, above problems cannot occur in DPL. Of course, changing a policy requires regeneration of the EECs, but since that happens automatically, this does not burden the user. We thus conclude that DPL is a fluid language.

Visibility and Juxtaposability

Visibility expresses how easy it is to access a given component. Ideally, this should be possible with a minimal number of steps and especially without cognitive work. An important component of visibility is juxtaposability, which simply means being able to display any two parts of the program side-by-side.

Since virtually any modern text editor supports viewing files side-by-side, juxtaposability is given for all three languages. In addition, for DPL, we support displaying the ontology next to the policy (see Figure 5.13). Since policies are atomic and do not have outside dependencies, DPL has high visibility. The user simply does not need to find any other component. In contrast, Java programs are usually distributed across many files, which are filed in elaborate package/folder structures. However, Java development environments are aware of this problem and thus support users with various ways to find a class. For example, users can type its partial name and displaying a matching list or open a class's/method's definition from a place where it is used. The only searches that cause cognitive load are of the kind "I know there is a class that does x, but I do not know its name". EPL again suffers from poor development environment support. There is no way to open the definition of a stream from a place where it is used. This might lead to a search for the right file, where it is defined.

Summary of Applying Cognitive Dimensions

We summarize our findings in Table 5.5. We can conclude that DPL is indeed easier to use than EPL or Java. This should not be a total surprise though, since the expressive power of DPL is much less than that of EPL or Java, which both provide much more fine-grained control. Yet we find above analysis useful to ensure the DPL aligns with our original goals.

Dimension	DPL	EPL	Java
Abstraction Gradient	+	○	○
Closeness of Mapping	++	○	---
Consistency	-	+	○
Diffuseness / Terseness	++	-	-
Error-proneness	○	-	-
Hard Mental Operations	○	-	○
Hidden Dependencies	++	---	○
Premature Commitment	-	○	---
Progressive Evaluation	-	+	○
Role-Expressiveness	++	+	---
Secondary Notation and Escape from Formalism	++	++	++
Viscosity	++	-	++
Visibility and Juxtaposability	+	-	+

Table 5.5.: Summary of cognitive dimensions analysis

5.6.3 User Study: Understandability of Code

Above analysis shows that DPL is easier to understand, especially for those not that familiar with programming. The analysis is, however, theoretical and needs some grounding. Thus, we conducted a user study among computer science students untrained in using event-based systems. We chose computer science students, because we wanted to include Java as a comparison language. We believe that this only makes sense if participants have at least some basic understanding of programming.

Setup

The study consisted of an online questionnaire in which participants were given a piece of code and presented with a number of statements about the behavior of this piece of code. They had to select whether they believe the statement is true or not. After that, participants were asked to specify on a scale from 1 to 5 how easy it was for them to answer above questions. We presented code and questions for the two scenarios introduced in Section 5.6.1, where the security scenario included 14 statements and the logistics scenario had 10 statements. We always presented the security scenario first. Each participant was randomly assigned to one of the four languages: Java, Esper EPL, Oracle EPL, or DPL. In addition to the correctness and the easiness of answering the questions, we measured the time it took participants to answer the questions. We eliminated outliers when comparing the response time⁷.

⁷ We observed response times of up to one hour, suspecting that participants interrupted answering the questionnaire and returned later

Results

We obtained a total number of 58 responses, out of which 18 were incomplete, leaving us with a total of 40 valid responses. If only one of the statements was not marked as correct or incorrect, we accepted the response and left the unmarked statement out of any calculations. Out of the 40, five responses had one statement not marked, albeit a different one each time. One response did not answer the question asking how easy it was for the security scenario. We also removed that response from the answers to this question.

Correctness

Figure 5.16 shows the results for the correctness as the average of how many percent of the statements were flagged correctly. Generally, when presented with DPL, participants were able to better understand what the code does. Oracle EPL and Esper EPL are similar in this category, which does not surprise given their similar syntax. We are a little surprised to see Esper EPL score slightly better than Oracle EPL, as we would have imagined the pattern syntax of Esper EPL to be less clear than that of Oracle EPL. This may, however, be clouded by pre-formulated statements, where participants simply need to make a true/false call and not figure out the exact meaning of code. Interestingly, Java scored worse than the EPL languages on the security scenario, but better in the logistics scenario. We think this is due to the much reduced code size in the latter scenario, allowing participants (who have all worked with Java before) to get a better overview. This observation is a good indicator that terseness and visibility are important cognitive dimensions.

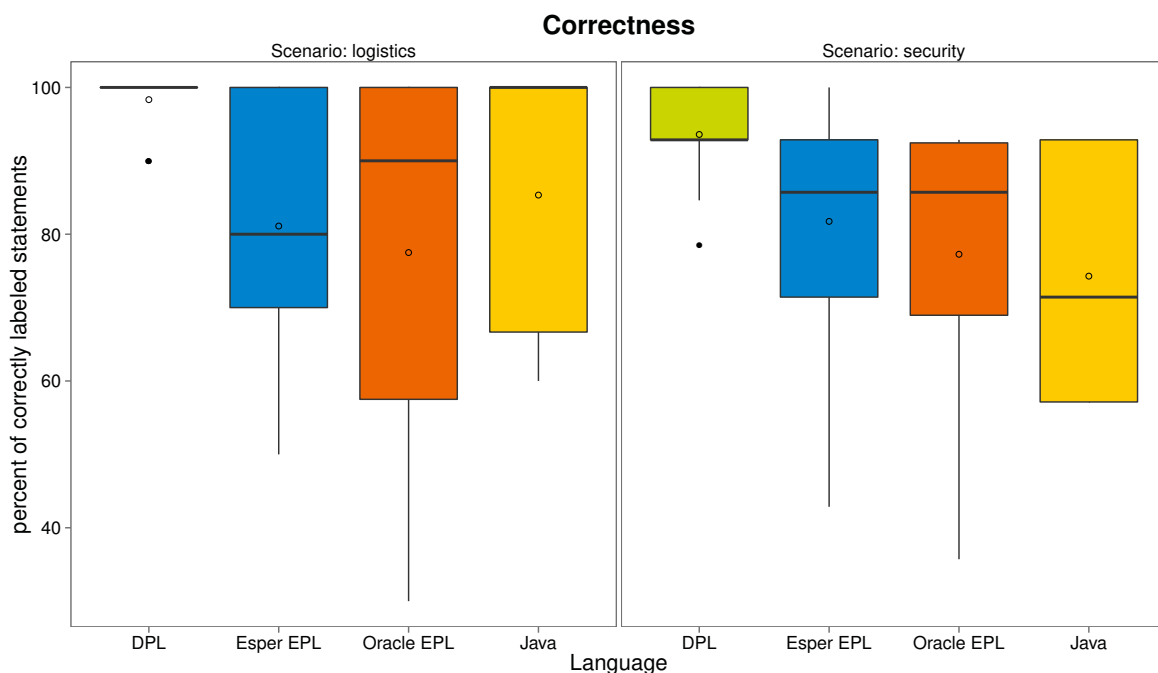


Figure 5.16.: Correctness results of the user study

Easiness

As Figure 5.17 shows, DPL was by far the easiest language to understand. Participants felt that answering the questions was somewhere between easy and very easy. We also see some indication that complex

boolean expressions are indeed a problem, as the security scenario contains one, whereas the logistics scenario does not. The other languages are much closer together, with a slight advantage for stream-

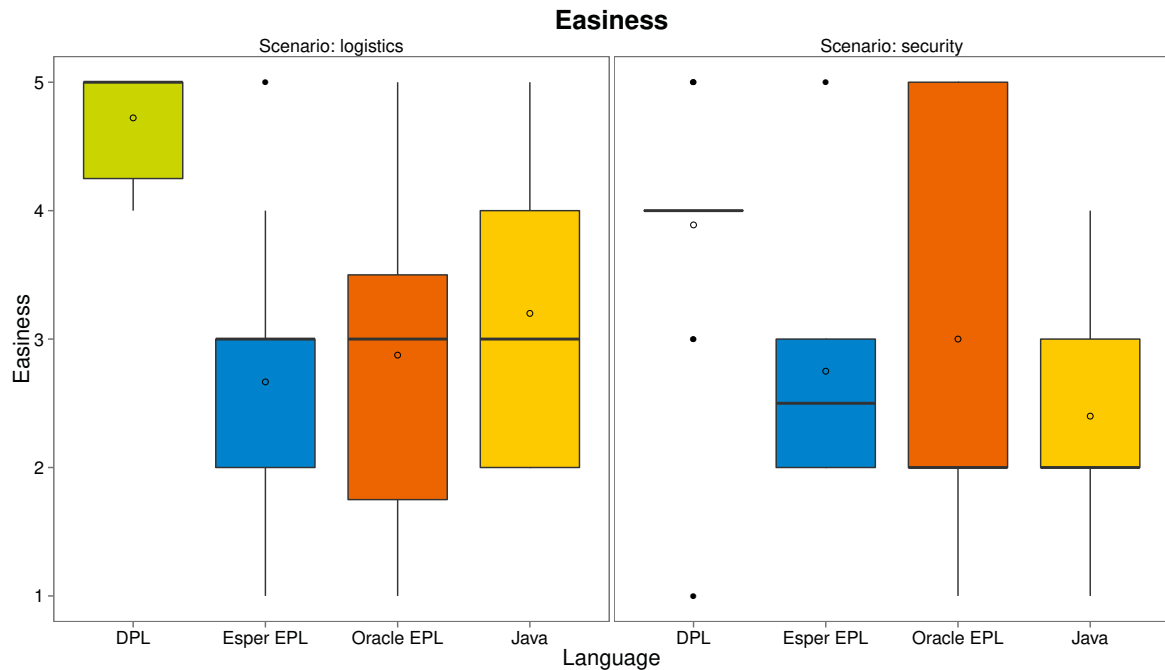


Figure 5.17.: Perceived easiness of labeling the statements; scale extends from very hard (1) to very easy (5)

based languages for the security scenario and a slight advantage for Java for the logistics scenario. Again, we think Java does better on the logistics scenario, because of the reduced code size.

Time

The average time it took to answer the questions is shown in Figure 5.18. Generally, it took less time to answer the questions of the logistics scenario. Even when comparing the average response time per statement, participants were still faster on the logistics scenario. We believe that this is due to two factors. First, participants were already used to the modus operandi in the logistics scenario and there was a little less code to scan. Results are what we expected. DPL is easy to understand without much effort and thus requires the least time to make qualified judgments, since it also scored best on correctness. Likewise, Java requires some serious cognitive work to see how the individual parts interact with each other, which is completely unsuitable for novice users.

Summary of the User Study

We conclude that DPL is indeed very suitable for its targeted application. It is considerably easier to understand than existing approaches, resulting in much reduced mental effort. In addition, the reduced complexity allows users to make more precise statements about the functionality of a piece of code.

5.6.4 Performance Evaluation

We present our performance evaluation results in this section. We start by describing the setup we used for the measurements and follow up with presentation and discussion of the results. We conclude the performance analysis with a brief summary.

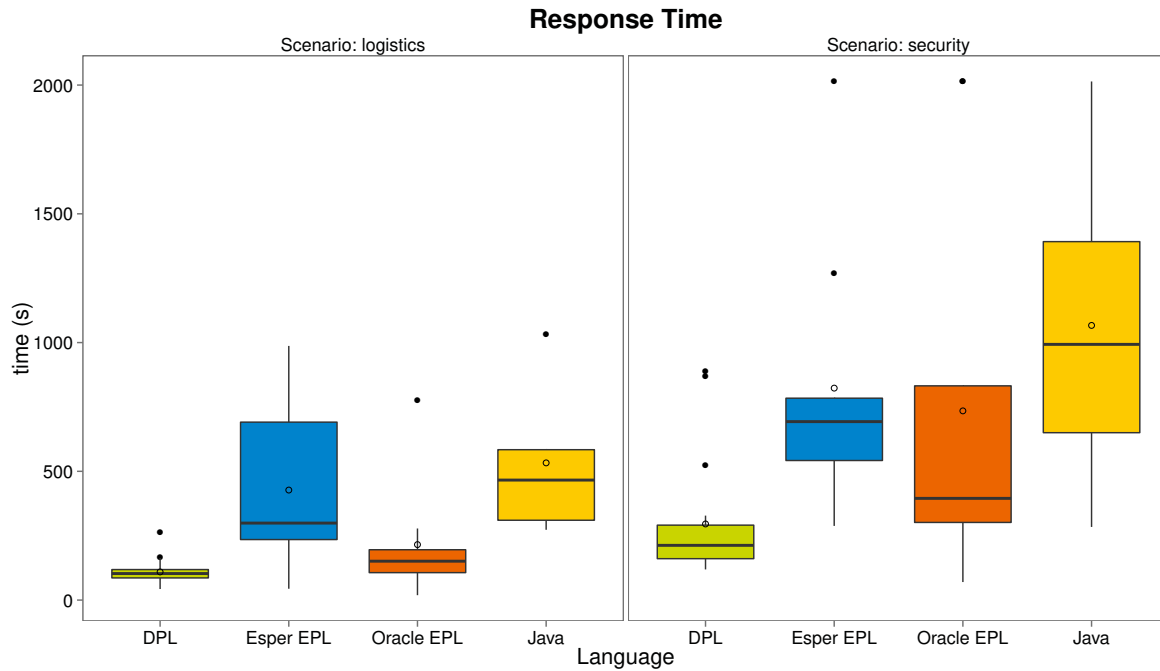


Figure 5.18.: Average response time for answering the questions

In the performance evaluation, we compared three approaches: DPL, DPL with sockets and Esper. DPL represents our approach as introduced in this section. DPL with sockets is the same approach, but uses sockets for inter-EEC communication as described in Section 5.5. For simplicity, we refer to this approach as "sockets". Finally, Esper represents the setup where we use Esper instead of our middleware to process and check events.

We measured CPU utilization and achievable throughput for these approaches. We did not measure latency, as the window sizes for detecting NOT-events and correlating events are order of seconds, whereas latency typically ranges in the order of milliseconds. Thus, any changes to latency would be masked by the window.

Scenario

We use our running example as the scenario for this evaluation. We used a smart home simulator created by Florian Reimold during his Bachelor's thesis [159] to model an event sequence corresponding to people moving around rooms (see Figure 5.19). Each room has a number of sensors which report positions of people moving through their detection area.

The simulator outputs an XML file containing the events and their occurrence times. We process this XML file and pre-generate event objects from it. During the experiment, we send events as JMS messages to the broker. If we reach the end of the event sequence, we just start at its beginning again for as long as the experiment runs. To generate different event loads while preserving semantics, we divide the time between events by different numbers.

Setup

We use five identical servers for our performance evaluation connected via a 10 GBit Ethernet (cf Figure 5.20). Each of the five machines serves a different role. The Workload Generator is responsible for

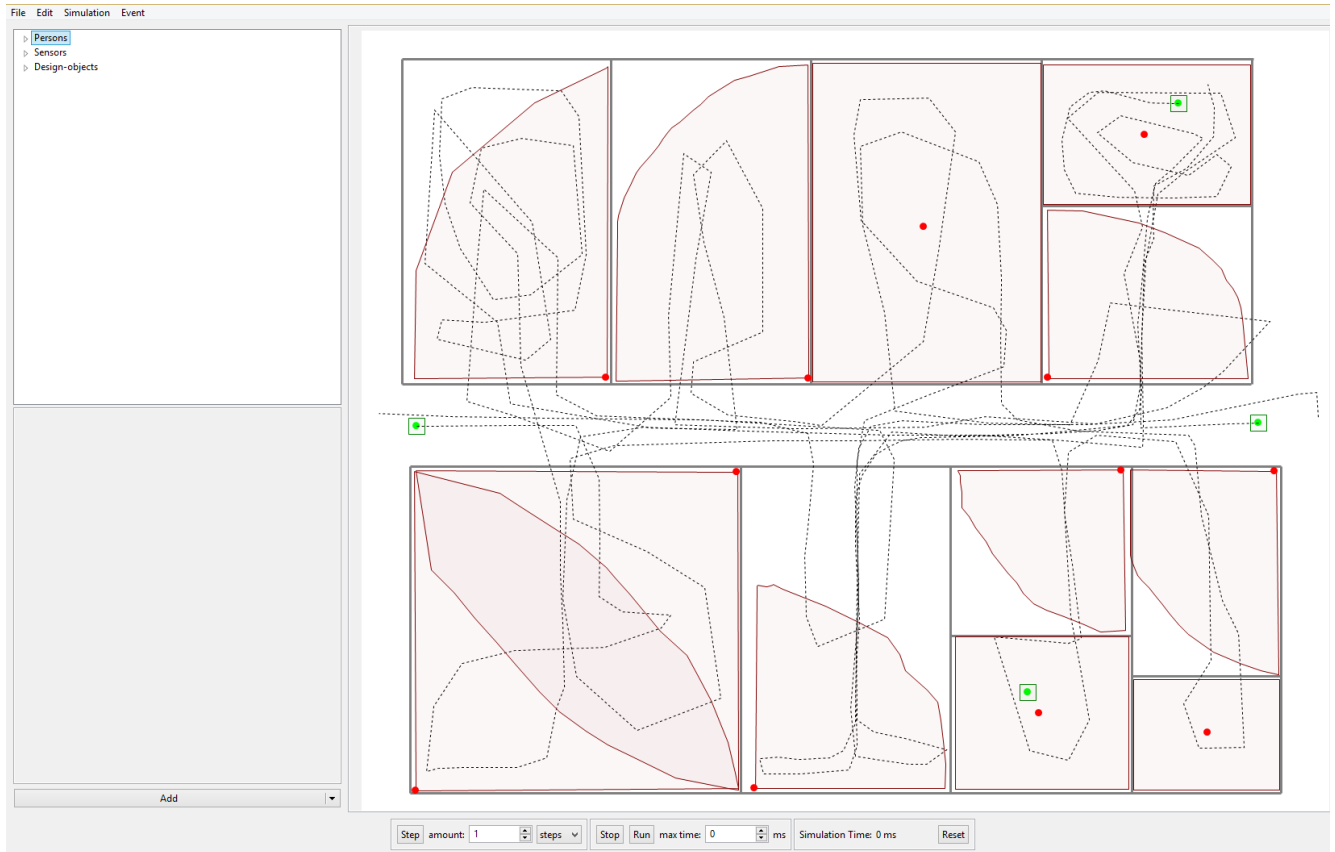


Figure 5.19.: Simulation to create the event sequence

sending raw events to a certain topic. The ActiveMQ Host acts as the publish/subscribe broker and runs ActiveMQ. The Policy Bootstrapper sends the policy to the middleware and monitors its results. The Experiment Controller coordinates all other servers, starting and stopping the appropriate processes on them. It also collects the results. The Contextualizer is the machine running our middleware and also hosts the generated EECs. When running Esper, we run the Esper code on this machine.

Each of our experiment runs has three phases: setup, running and drain. During the setup phase, we start the individual components on our servers, giving each component enough time to setup. In the running phase, we use our event generator to generate events. Note that while we set different event rates for the entire duration, the event rate is not constant, because sometimes multiple sensors report at the same time. We use the drain phase to allow for any delayed events to arrive and other resources (i.e. the database) to answer any pending requests so they are not occupied when the next experiment starts. Our running phase lasts for five minutes; our drain phase lasts for two and a half minutes.

CPU Utilization

We measured CPU utilization every second across all available cores. Our servers have 16 cores which are capable of hyperthreading, resulting in 32 visible cores. Before we started the measurements, we observed the system while it was working using the monitoring tool `htop`. We found that our processes put load only on the first 16 cores. Thus, when computing overall CPU utilization, we averaged the utilization of the first 16 cores and not all 32. We only took the first five minutes into account, as this corresponds to our measurement phase. Since the CPU was mostly idle during the drain phase, we excluded this phase from the measurements, as we feared it might skew results.

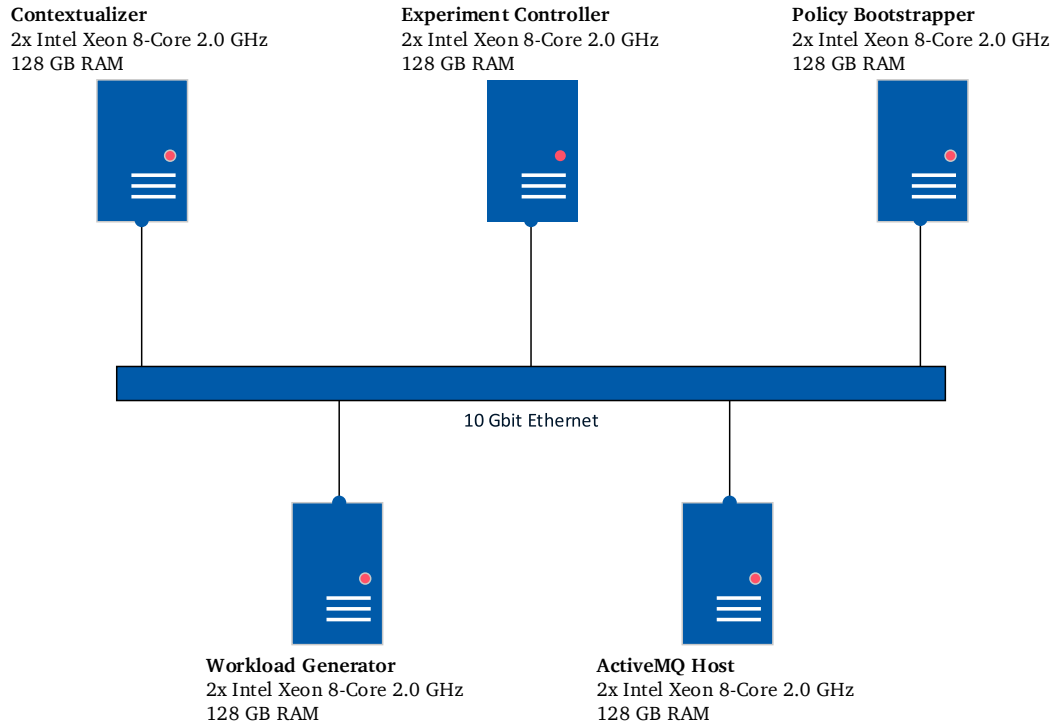


Figure 5.20.: Setup for the performance measurements

We measured CPU utilization on the Contextualizer server and how well that load distributes across the available cores, which is an indicator for the parallelizability.

Single Workload

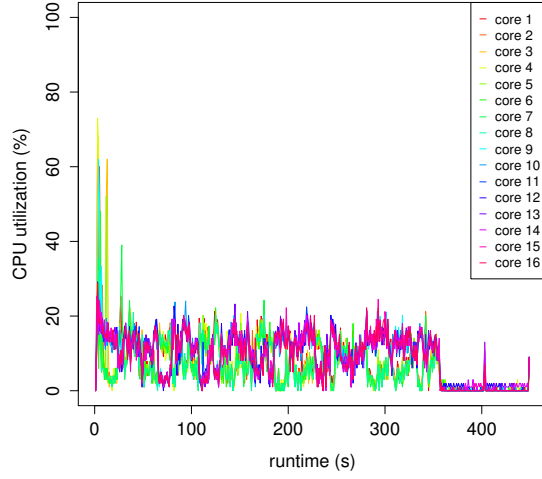
Figures 5.21a - 5.21c show the utilization of each core over the duration of the experiment. In experiments using our approach, utilization of the cores is much closer together than in the Esper scenario, which has a higher utilization on fewer cores.

Figures 5.21d - 5.21f illustrate this behavior further, by applying statistics functions to the utilization of the cores every second. The average utilization is shown in Figure 5.21d. It appears that Esper has a generally lower CPU utilization than DPL and Sockets, which perform very similarly. However, the maximum utilization of individual cores is much higher in Esper (Figure 5.21e), which is also indicated by the standard deviation of the CPU utilization of the cores (Figure 5.21f).

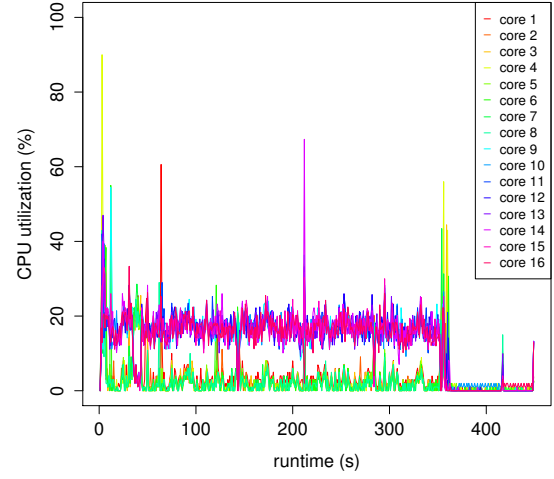
This illustrates the parallelizability of our approach. The generally higher CPU utilization can be explained by a greater number of serialization operations, as more messages are sent compared to Esper which computes everything on one machine. Furthermore, Esper is a highly optimized product, while we used our research prototype. However, Esper is not able to utilize all available cores, while we uniformly distribute load across the cores, because we have individual EECs working together.

Average CPU Utilization

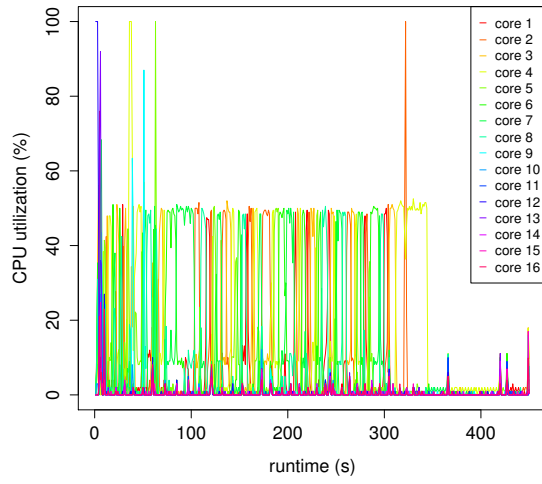
After analyzing a single experiment, we compared the CPU utilization using different loads. Figure 5.22 shows the results of this comparison. For Figure 5.22a, we averaged the CPU utilization of all cores every second and then averaged these values to get the total average CPU utilization for the duration of the



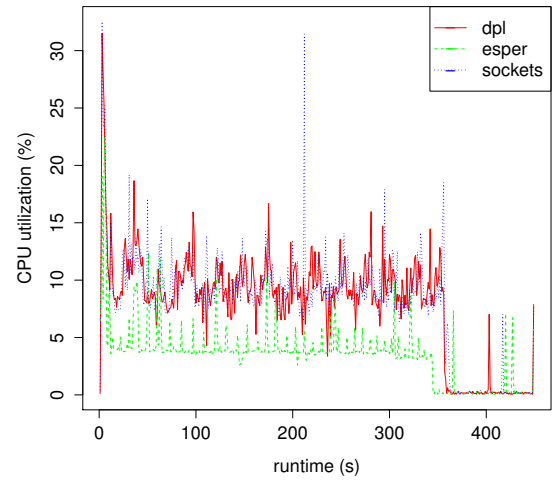
(a) DPL CPU utilization



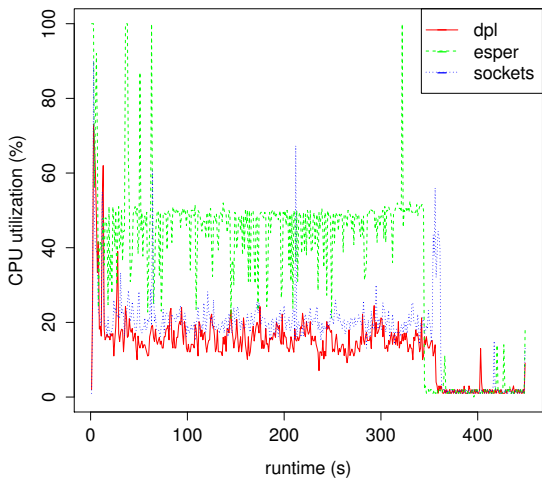
(b) Sockets CPU utilization



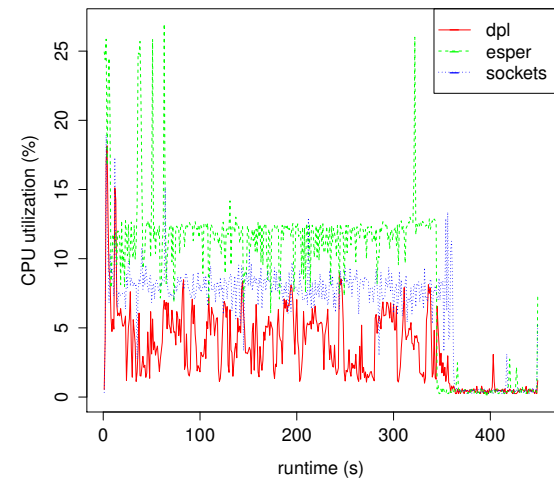
(c) Esper CPU utilization



(d) Average utilization



(e) Maximum utilization



(f) Standard deviation of utilization

Figure 5.21.: Utilization over time for 1300 events / second

measurement. The figure shows that our approach causes more CPU utilization than doing the processing with Esper does. We are not surprised by this, since Esper is a highly optimized, industry-ready product, while our implementation is still a prototype.

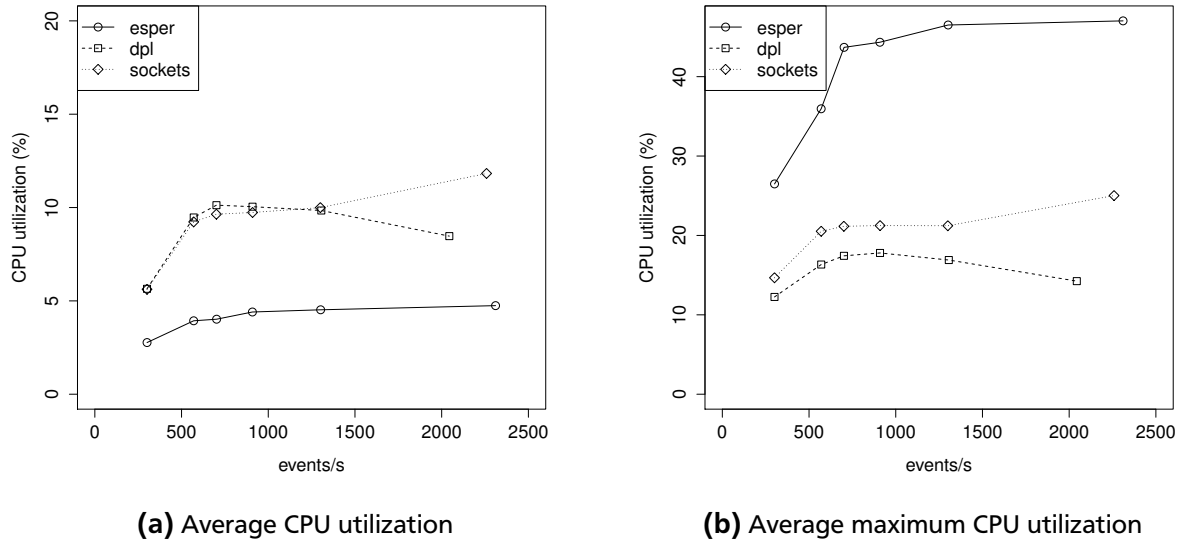


Figure 5.22.: CPU utilization of the Contextualizer

Figure 5.22b confirms what we observed above. Instead of averaging the CPU utilization across all cores, we computed their maximum value each second and then averaged these values, to get the average maximum CPU utilization. We did this to get an indication of how much of the work is happening on a single core. As the figure illustrates, Esper has much more single-threaded parts than our approach. Figure 5.23 confirms this observation in consensus with above. We see this as an indication that our approach is much more parallelizable than Esper.

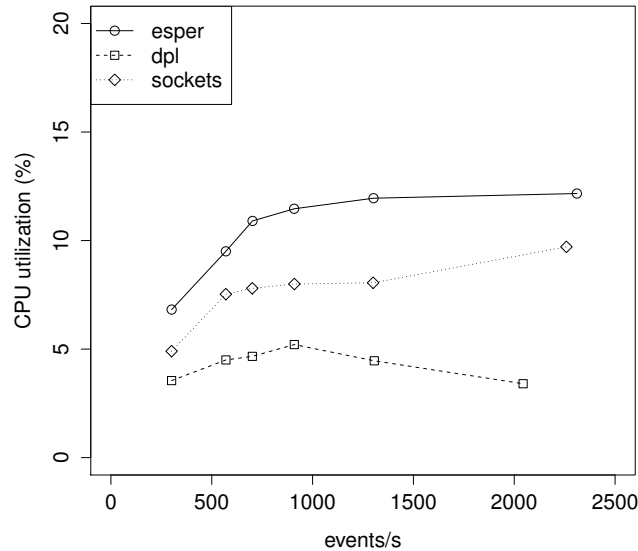


Figure 5.23.: Standard deviation of the CPU utilization

The decrease in CPU utilization for the DPL case is subject to throttling of ActiveMQ. When ActiveMQ notices that a client is not able to process messages as fast as it sends them, it automatically decreases the message rate. The reason of the throttling despite a relatively low CPU utilization is a self-energizing effect: there are two operations on the client which take time but do not require much CPU power. These

are accessing the database and sending a message. When the rate of messages sent is momentarily too high so that the client can process them in time, ActiveMQ will throttle the messages to this client and soon after also slow down producers so messages do not queue up on the broker. However, sending a message is part of handling a message in EECs. Thus, if the sending of a method takes longer because of throttling, so does the handling of a message, which in turn leads to more throttling. As the figure shows, using sockets instead of JMS for inter-EEC communication does not suffer from this effect, despite doing the same work.

Figure 5.24 shows the same measurements on the ActiveMQ host. They confirm our observations above. Average CPU utilization increases with load, until throttling kicks in for the DPL case. The average maximum CPU load is nearly the same across all scenarios, albeit slightly higher for our approach, because there are three consumers for the primitive events (the leaf-EECs) instead of just one as in the Esper case.

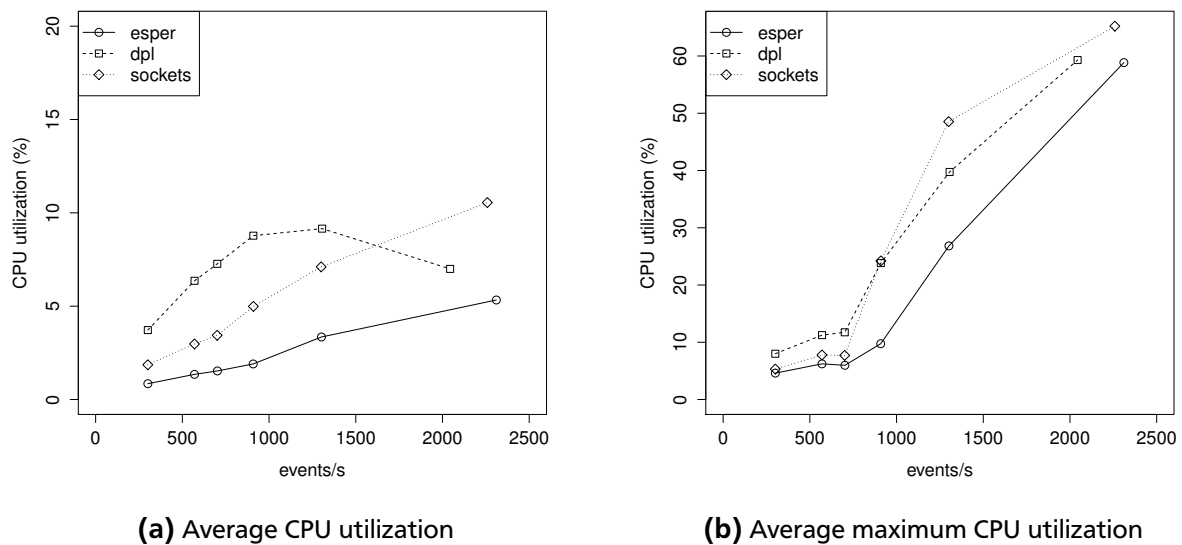


Figure 5.24.: CPU utilization of ActiveMQ

Throughput

Figure 5.25 shows the maximum achievable throughput for each scenario. We obtained the numbers by counting each message sent on the producer side. Since we did not observe broker or contextualizer activity at the end of the drain phase, we are certain that events did not queue up at the broker (see Figure 5.21). Thus, counting at the producer side is a reliable metric.

The figure confirms the indications of the previous section: the throughput of our approach is less compared to running Esper. However, Esper is a highly optimized system, whereas we use a research prototype. In addition, the goal of our approach was not to improve event processing performance, but rather to make it more usable. We are thus not surprised to see a drop in performance.

We suspect that the sending of messages has a huge performance impact. The evaluation of ACTrESS (see Section 4.4) showed that the marshalling and unmarshalling of messages has a performance impact. Since we need to do this even when using sockets, we also suffer from the performance impact. Esper does not support distributed computation and sends events internally, which allows for avoiding (un)marshalling.

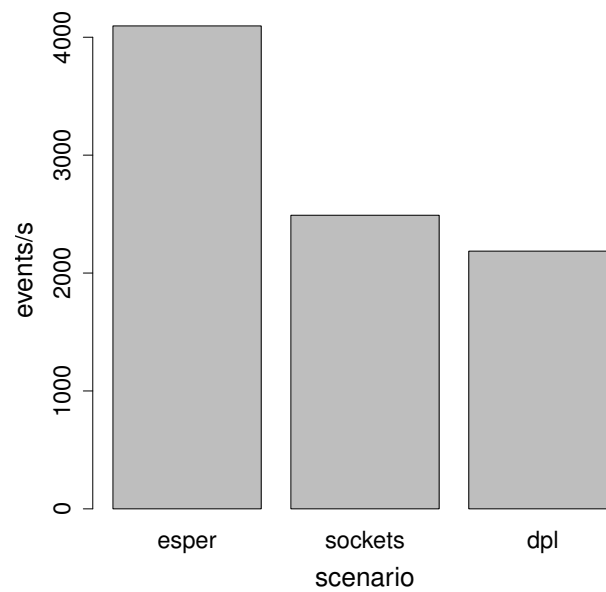


Figure 5.25.: Achievable throughput

5.6.5 Summary of the Evaluation

We evaluated usability and performance of our approach. The performance analysis shows that our prototype implementation can handle fewer events per second and incurs a higher CPU utilization. We mainly suspect the comparatively high amount of (un)marshalling operations as the main reason. In combination with ActiveMQ's producer flow control, this leads to a reduced number of messages sent. Our approach uses the available cores in a much more even manner, indicating that it indeed allows for good parallelization.

The theoretical and practical usability analyses show that our approach and its underlying language DPL are much easier to use than Esper or Java. This is mostly due to the reduced complexity and a very good alignment with the mental model of the user, which also makes our approach less error-prone. Thus, despite a performance disadvantage, we achieved our goal of simplifying the use of Cyber-physical Systems.

Part III.

Related Work and Conclusion



6 Related Work

You never change things by fighting the existing reality. To change something, build a new model that makes the existing model obsolete.

–Buckminster Fuller

In this chapter, we present our analysis of related work. Following the structure of the thesis, we separated discussion of related work into *Contextualization* and *ACTrESS*. Neither the idea of event transformation nor the idea of enrichment/contextualization are radically new ideas. In their book *Event Processing in Action* Etzion and Niblet suggest that any processing of events is performed by a network of Event Processing Agents (EPAs) [74]. An EPA may be a transformation agent. Transformation agents may perform one of six possible transformations: translate, compose, aggregate, enrich, split or project. ACTrESS supports all but splitting an event. Our contextualization process makes use of enrichment. While Etzion and Niblet provide a good framework for describing event processing systems, they do not give any information on *how* these EPAs work. We have given concrete mechanisms and implementations in this thesis.

6 ACTrESS

In our analysis of related work, we identified two main categories of related approaches: contextual mediation in publish/subscribe systems to improve matching algorithms, and integration of different data sources. The latter has been researched extensively in the data integration community, but we will give related work only on selected aspects, as our focus is on contextual mediation in EBSs.

To get a better overview of related work and avoid an unstructured listing of approaches, we grouped related work into several categories: *Semantics in events* refers to approaches that encode the semantic meaning in the event while *semantic metadata and subscription matching* focus on the matching algorithm. *Semantic web* groups the approaches that rely on ontologies for semantic interoperability. *Integration solutions and EAI* encompass famous integration principles and software. Finally, *database integration* groups integration techniques that are used for databases. While we do not claim an exhaustive collection for each category, we chose to list representative or important work.

Existing approaches fall short on one or more of the requirements outlined in Section 4, which we briefly summarize below. Since shortcomings are shared by each category, we aggregated our findings per category (see Table 6.1).

Anonymity

indicates if the approach allows endpoints to be anonymous to each other.

Dynamism

means the support of joining (and leaving) endpoints at runtime.

Low latency

shows if solutions support low latency transmissions.

Flexibility

indicates which approaches do not rely on an a priori agreement to one global (immutable) schema.

Transformation support

means that the approach supports expressive transformation specifications beyond providing just an API.

In addition to these requirements, we also analyzed related work according to our partitioning of the transformation space (cf. Section 4.1.3). If an approach does clearly not support the highest level in a category, we indicate this by stating which level the approach supports.

	Anonymity	Dynamism	Low latency	Flexibility	Transformation support
<i>Semantics in events</i>	✓	✓	~	~	
<i>Subscription matching</i>	✓	✓		✓	
<i>Semantic web</i>	✓	✓			
<i>EAI</i>	~	✓		✓	
<i>Integration solutions</i>			~	✓	✓
<i>Database integration</i>					✓
<i>ACTrESS</i>	✓	✓	✓	✓	✓

Table 6.1.: Feature comparison. ✓ refers to supported features, and ~ to weakly addressed requirements.

Semantics in Events

Several researchers identified the need to consider semantics when dealing with events. Cilia et al. suggest that the integration of new clients into an EBS in presence of data heterogeneity requires mediators [52] and “*explicit information about the semantics of events*” [53]. They advocate enhancing the notification mechanism by allowing producers to pass semantic information and consumers to receive data in their semantic context. To exchange semantic information, they use a self-describing model described by Bornhövd et al. [27] (E3). This requires explicit handling of context by client components and substantial additions to core middleware interfaces. While their ideas have influenced our work, we believe we improved on many aspects. Our approach stores context information inside the broker network for performance reasons (see Section 4.4) while remaining modular and flexible. Furthermore, we show that our approach can be integrated into existing, industry-strength software and provide a performance analysis.

Singh et al. suggest a policy-based approach to control the flow of data in publish/subscribe systems [186]. The policies they suggest can be used to deny specific events to subscribers or transform events (based on filter criteria) before and after subscription matching. Transformation functions may delete or change certain attributes of an event, but are always defined for entire event types (G1).

Similarly, Wun and Jacobson introduce a framework for using policies in content-based publish/subscribe systems [214]. Policies are applied upon the matching of an event to a subscription, though policies may also trigger before the matching. The work is focused on the management framework of the policies and does not detail the application of said policies. While the authors mention message transformations as one example of a policy action, they do not detail how such transformations are defined and assume the existence of universal transformation functions. Thus, from a client’s perspective, transformations are defined on the entire message content (G1).

Scherp et al. [175] introduce an event model which supports different event interpretations (G1). They approach semantics from another angle and argue that the same set of events can have different interpretations and causality, depending on the context of the observer. The work is conceptual and focused on developing a sound event model. Thus, they do not provide mechanisms to transform between interpretations.

HydroJ [127] is a system, where components exchange self-describing, semi-structured messages. Messages are handled at the receiver side through *handlers* (T1). Handlers use pattern matching to specify which messages they handle and what kind of messages they return. They can thus be used to transform *inessential* message parts (e.g., format, data types) to the receiver's needs (E1). This enables components to evolve independently. Transformations occur on the whole message body (G1).

Acute's Hashtypes [183] hash type information and function signatures to share them the same way the contents of messages are shared. Thus, receivers get information about the types the sender used, supporting heterogeneity to a certain degree. However, Acute is focused on point-to-point communication and thus transformations are applied to event objects at the receivers (T1). Value-based transformation is not supported (E1).

On a more structural level, the idea of *structural subtyping* proposes to use only the structure of objects (and not the name) to define subtypes [40]. An object type T_1 is a subtype of T_2 , if T_1 has more fields than T_2 and the common fields have compatible types. This is especially helpful in distributed settings with potentially different programming languages involved. Lingua Franca builds on that idea and defines a domain specific language for defining object interfaces in distributed object systems [141]. Unlike in most modern programming languages, an object's interface is not defined by a set of identifiers, but by the set of methods it exposes. Requirements to parameters are specified by a set of methods that must be supported. This allows for conformance checking without an a priori definition of namespaces. Acute also suggests structural subtyping [183]. Some publish/subscribe brokers like Siena [42] or JEDI [59] use structural subtyping as well. Compound Types suggest using the flexibility of structural subtyping in combination with the expressiveness of name equivalence [34]. Programmers can express that they need the compound of two (or more) interfaces to be supported, but the actual checking happens on a structural level. Whiteoak [99] builds on this approach and provides an extension for Java for structural typing. The authors solved many issues like recursive types and are thus able to provide a fully working compiler for their language. All of the above approaches operate purely on a structural level and do not offer support for value transformations (E1).

Semantic Metadata and Subscription Matching

Some related work addresses the field of matching metadata and subscriptions semantically. Ruotsalo and Eyvönen [170] argue that different metadata schemas hinder interoperability and promote the idea of transforming individual schemas into a shared representation (G1, E1, T2). Their work focuses on the development of mapping rules.

Skovronski and Chiu describe a publish/subscribe framework which uses semantic data to improve expressiveness of the subscription language ("ontology-based publish/subscribe") and aims at making subscriptions interoperable [188] (E2). They introduce state at the broker, enabling the matching to depend not only on the current event but also on previously received events. While they argue that performance and scalability are factors to consider, their approach updates the ontology model for each incoming event and then runs subscription queries against the ontology. The drawback of this approach is that their system takes multiple seconds to process a single message.

Similarly, the Semantic Toronto Publish/Subscribe System (S-ToPSS) [39, 153] uses ontologies to match message data to subscriptions, making use of synonyms, relationship knowledge (specialization and generalization), or schema mapping functions (E2). The authors do not rely on a single, global ontology, as they believe that there will be multiple, domain-specific ones. However, they do not provide details on how mappings are defined and executed, nor a performance analysis. Mappings seem to be defined on the whole message body (G1).

Wang et al. introduce a publish/subscribe system where subscribers express their interest in events in the form of graph patterns [210]. The Resource Description Framework (RDF) is used to represent events, converting incoming events into RDF automatically (G1). However, events are delivered to the subscribers always in RDF. Thus, subscribers cannot get events in the format they desire and might still have to do their own interpretation, with the additional information encoded in the RDF representation (T1). The focus of their work is on developing a more expressive subscription language including correctness proofs. While they do provide a performance analysis of their system, they do not provide any information about the CPU load their additions cause.

All above-mentioned approaches use expensive transformation mechanisms, which have a negative impact on latency. Furthermore, they focus on the matching, without providing events to consumers in their interpretation.

Semantic Web et al.

The semantic web [18] shares some goals with ACTrESS, specifically to express the meaning of data. Knowledge is often represented with ontologies; for example both an `XYPosition` and a `LatLonPosition` are a `Position`, and thus conceptually the same thing. However, semantic web techniques approach the problem from a different angle: ontologies express the existence of a relation explicitly, but the exact incarnation of that relation only implicitly through rules, if it is specified at all. In the present setting we are inversely interested in explicit low-latency transformations between say `XYPositions` and `LatLonPositions`, without necessarily putting them into any relation.

This distinction also becomes apparent in research conducted by Blair et al. in the context of the European research project `CONNECT` [17, 23, 101] and preceding work on reflective middleware [22]. Their approach uses ontologies to establish interoperability in heterogeneous, distributed systems, by providing automatically generated connectors. Heterogeneity is addressed with regard to data (syntax and semantics) and application behavior. Participating systems, which they call *networked systems*, add semantic information to the advertisements about the services they offer and to the requests for services they want to consume. *Discovery Enablers* collect this information and use machine learning techniques to match service requests with offered services. To that end, they generate adapters which mediate the data exchange between matched services. We see their work as complementary to ours in three ways: (1) *Scope*: the focus of `CONNECT` is to mediate between services with heterogeneous behavior; ACTrESS in contrast is designed for message-oriented system federations and puts the focus on mediating the heterogeneity of the data exchange, not application behavior. (2) *Complexity*: `CONNECT` employs machine learning techniques and makes strong use of ontologies to discover the behavior of systems, while ACTrESS provides lightweight context descriptions, optimized for low-latency transmissions. Ultimately, our context descriptions could be a result of `CONNECT`'s elaborate process. (3) *Architecture*: In `CONNECT`, the *Discovery Enablers* use hard-wired plugins for each protocol, to understand data in advertisements and service requests; ACTrESS in contrast gives endpoints a means of expressing their interpretation of data and transforms subscriptions and notifications transparently.

Wache and Stuckenschmidt [194, 208] describe a model for context transformation for semantic interoperability between different information sources. Their approach uses description logic to describe ontologies for the specific application domain, called the *shared vocabulary*. This vocabulary is established for all information sources, before performing transformations. The transformations themselves use lookup mechanisms (E2). Furthermore, they suggest two different kinds of contextual transformation – rule-based functional transformation and classification-based transformation – and give a unifying model. Wache and Stuckenschmidt focus on presenting a formal model for context transformation, which they prove to be correct and complete for their example domain. They integrate their approach into a system, which operates on a centralized database (T2).

Breitman et al. suggest using ontologies to make the service descriptions in a multi agent system interoperable [29]. Therefore, agents relate the descriptions of their service specifications to a common ("upper") ontology. They claim having created a single reference model for multi agent systems, but argue that in future work, they want to investigate ontology alignment techniques. Even then, agents themselves are responsible for transforming data (T3). No details on the actual transformation mechanism are given.

Enterprise Application Integration (EAI)

EAI specifies *message transformations* [111] to deal with heterogeneity. However, EAI just specifies a pattern as an architectural suggestion, which can be seen as a more detailed definition of adapters [94, 138], without suggestions for its implementation. EAI frameworks like Apache Camel [114] thus support message transformations, but provide merely an API without support for implementing it. The entire message body is subject to the transformation (G1) and there is no support for constructing a *canonical data model*, leaving its design to the programmer. As demonstrated in Section 4.4, efficiency is poor.

Oki et al. specified constraints and architecture for a large-scale, distributed system, which they call the *Information Bus*. The Information Bus uses self-describing messages like some of the approaches above, but is mainly focused on integrating existing systems via adapters. Thus, we see it as early work on EAI.

Although not originally designed for publish/subscribe systems, Java Internationalization (JI)¹ in combination with Java Remote Method Invocation (RMI)² (or similar remote communication paradigms) provides a good foundation to heterogeneity in event-based systems: JI provides different interpretations of data depending on the context like different character sets for strings, date conversion between timezones. JI can translate between different natural languages automatically, by looking up strings (e.g., entire sentences) in dictionaries (E2). *Resource bundles* are application-specific types, which JI automatically mediates across different *locales*. However, JI is mostly just an API, with little support of implementing the actual transformations. Our work in contrast provides intuitive and safe mechanisms for such an implementation. .NET Internationalization³ and other internationalization frameworks provide similar mechanisms for their respective platform.

Integration Solutions

General data transformation languages like XSL Transformations (XSLT) [54] can specify arbitrary transformations. However, they do not provide the structural support for organizing corresponding rules with

¹ <http://docs.oracle.com/javase/8/docs/technotes/guides/intl/>

² <http://docs.oracle.com/javase/8/docs/technotes/guides/rmi/>

³ <http://msdn.microsoft.com/en-us/goglobal/bb688096.aspx>

clear priorities into interpretation contexts and for extending these. Thus, we see these languages orthogonal to our approach: we could easily modify our system to generate XSLT code instead of Java code, which can then be used to transform Extensible Markup Language (XML) documents.

Integration solutions like OpenII [182] or Microsoft BizTalk [164] provide support for transformations, for example with XSLT and *orchestrations*. However, mappings are globally defined and thus new producers or consumers have to be added explicitly. Furthermore, mappings are static and defined on the entire message body (G1). In addition, mappings are always defined from endpoint to endpoint, creating the problem of a quickly growing pool of mappings. Limiting the mappings to communicating endpoints sacrifices the flexibility of publish/subscribe systems.

Cluet et al. argue that integrating heterogeneous data sources is often done with non reusable software components [56]. They suggest a declarative rule language for a mediator-based middleware. Rules consist of a body which specifies patterns and predicates to filter input data. The head of a rule describes how matching data is restructured (E1). Their approach is designed for a request/reply interaction and requires communication partners to be known.

The approach suggested by Foster et al. transforms tree-structured data between different views [85]. Thus, changes made in one view are reflected in all other views. The authors say that tree-structured data (e.g., XML) is a simple-enough assumption that almost any kind of data can be reflected by that. They rely on an intermediary data format, which they call the *common abstract view*. *Concrete views* then define a function to and from the abstract view (E3). We deliberately support unidirectional, flexible transformations as communication in publish/subscribe networks happens one-way and events are not shared like documents.

Dozer⁴ supports mapping of data objects between Java Beans. Dozer supports expressive and complex, albeit static, mappings; conversion resolution and execution occur via Java Reflection at runtime, limiting performance and safety. It is highly recommended to use Dozer as a singleton (T2).

Communication frameworks like Sun RPC [200], OMG's CORBA [103], or Web Services [6] only translate different *encodings* (e.g., little vs. big endian) but not structure or the values themselves.

Database Integration

Obtaining information from various heterogeneous databases is a similar problem to heterogeneity in event-based systems. Database integration is the art of transforming data from one database so it adheres to the schema of another database [3, 149] (T2). It has a long research history (e.g., Multi-base [65]) and led to many commercial products like Informatica®. However, *schema integration* requires a priori knowledge of the communicating endpoints, which is contrary to anonymity in publish/subscribe systems: a subscriber does not know the identity of an event's producer and thus cannot know the schema of that event.

We give only a few examples for this category, because the drawbacks when applied to event-based systems are the same across all database integration approaches. The approaches we selected, share some ideas with our approach, like transformation rules or the idea of contexts.

Bressan et al. [30] argue that a mediator should rewrite database queries and query results to abstract from different representations and units. Unlike many other database integration approaches, they do not rely on a priori defined mappings. Instead, they use *context mediators*. Queries are formulated

⁴ <http://dozer.sourceforge.net/>

against a common domain model and the mediators automatically resolve any conflicts between data-sources, which have to be explicitly listed as part of the query (T2). Gannon et al. [95] extend this idea by automatically generating *conversion programs* based on six parameterizable conversion rules. They use *contexts* as a container that describes a data source's interpretation of data. They too argue, that agreeing on a global schema or ontology is prone to fail and thus provide clients with a way of describing their local ontology. The local ontology uses the six conversion rules to define the transformations and mappings with which new information sources can be integrated. A mediator component uses this information to mediate queries through query rewriting (T2). While sharing some assumptions with ACTrESS, both approaches cover only pull-based, centralized database access, with known recipients of requested data.

Chung argues that agreeing on a single database management system (DBMS) is infeasible in large organizations, because individual business units have diverse requirements and existing databases and applications might cause a high conversion cost [51]. Chung suggests embracing the coexistence of different DBMSs and introduces DATAPLEX, a middleware system which manages the different DBMSs and enables uniform access to the data through a common relational model. Data mediation is done by a centralized components (T2).

Guo and Sun suggest a concept-centric approach to exchange product data between companies with focus on the domain of e-commerce data exchange [107]. They suggest assigning a context to each company, and transform between contexts when a company requests data. Their approach encompasses assigning concepts to product data, which are then transformed between contexts (G1,E2). Using contexts and concepts, they suggest creating XML product maps, which may then be queried by companies (T1). Thus they advocate a common concept in relation to which other concepts are defined, yet all participants must agree upfront on a global concept.

6 Contextualization

In our analysis of related work, we found that many researchers are aware of the complexity of Event-based Systems (EBSs). Thus, a large body of work tries to make their development easier. Usually this means taking a known concept such as Complex Event Processing (CEP) rules and improving slightly on that, for example, integrating the rule language into the host language. This does, however, not solve the problem of the inherent complexity of these rules. We also found that while policies or similar declarative statements are a popular concept for specifying some high-level rules, related work limits the use of their policies to a specific target domain (e.g., IT security), while we propose a much more general approach to stating reactive behavior. Other related work on reactive systems usually relies on Event-Condition-Action (ECA) rules to specify behavior, often without being aware of the introduced complexity.

We grouped related work into five categories. The *CEP languages* category looks at general language support for CEP; *Situational Awareness* groups work that approaches reactive systems from the viewpoint of a situation, to which reactions are defined. The category *Simpler development of reactive and cyber-physical systems* provides an overview of frameworks and middleware which support the development of Cyber-physical Systems (CPSs), while *Policies as declarative statements* explores related work using policies in some form. To keep the focus, we only regarded related work on reactive or event-based systems for this category. Finally, *Other related work* contains related work which did not fit in any of the other categories.

CEP Languages

CEP [132] is an active field of research. It is based on stream processing and active databases, aiming at providing higher-level abstractions with event patterns, filtering and aggregation. Low level events are processed and combined according to a set of rules, expressed in a CEP language (cf. Section 2.2.1). Eckert et al. surveyed existing CEP languages [71] and categorized them into five groups: composition-based languages, data stream query languages, production rules, timed state machines and logic languages.

Composition-based languages closely reflect operators of the underlying event algebra, showing their active databases [64] heritage. IBM's Active Middleware Technology [4] is a system using a composition-based language.

Data stream query languages express event compositions in SQL-like event stream queries. This style is the most successful one in terms of product support.

Production rules [83, 91] take a slightly different approach. They work over a set of *facts*, which reside in a *working memory*. Rules express conditions over these facts, and an action to take when the conditions are satisfied. A rule engine then incrementally evaluates the production rules [84]. Event processing can be realized by treating events as facts and producing facts/events in the rules' action part. However, as Eckert et al. state "In doing so, the programmer has much freedom but little guideline". Despite that, production rules have found their way into event processing software like TIBCO Business Events.

Timed state machines extend general state machines (also known as finite automata) with temporal aspects. Nodes represent the states of a system and edges between these nodes the events that cause state transitions. In timed state machines, the edges may also include temporal constraints. A particular state is reached by a well-defined series of events, which we may see as a form of a complex event.

Logic languages map event queries to the style of logic programming (e.g., Prolog [55]). An advantage of this approach is the strong formal foundation provided by logic programming. An early representative is Event Calculus [123], upon which most other approaches build.

All these five different styles operate on the same level of abstraction: programmers have to specify individual events and how they are combined. Only the means in which this intent is expressed differs. Thus, any of these styles is comparable to EQL in terms of complexity.

With CEP becoming popular in the industry, a plethora of complex event processing and stream processing have emerged [60]. Most research in this area, however, aims at improving performance [212] or increasing the expressive power of the rule language [67]. However, all these systems do not aim at lowering the general complexity these systems yield. This has been acknowledged by other researchers as well [81], but rarely tackled yet.

There is some research concerned with tackling the complexity of event-based systems. For example, Schiefer et al. developed a graphical tool for specifying event-condition action rules [176]. Their industry-ready system displays event conditions as graphical entities. Programmers connect different event conditions, by graphically connecting their offered ports to incoming ports of other event conditions (e.g., a preconditions port). Examples for ports are true/false ports of conditions or a matched port for pattern definitions. Thus, programmers can create graphical layouts of their event processing rules. While this approach certainly helps in understanding a system's behavior by visualizing event flows, it still operates on single events and programmers need to know the concepts behind complex event processing. Using graphical notations is also the direction that major commercial CEP engines go. For

example, TIBCO's StreamBase CEP [201] represents CEP queries as a series of graphical operators, between which streams of events flow⁵. While this provides a good overview of the rules themselves, a lot of details are still hidden in the configuration parameters of the operators. Furthermore, this approach only changes the language in which CEP rules are expressed from textual to graphical, but operates on the same level of abstraction and requires expert developers to use appropriately. Other commercial products like IBM WebSphere Business Events [113] or Oracle Event Processing [148] rely on similar graphical notations, which are, however, less graphical and more textual.

There are many extensions to existing languages which introduce an event-based style to support a specific application domain like context-awareness [126], layer transitions [116, 118], or monitoring [122]. All these extensions however, are limited to one application domain and the built-in support may simplify some things, the abstraction level remains on dealing with single events.

Ethion and Niblet suggest applying the notion of context to event processing [74]. While they do not provide a concrete language, they introduce mechanisms that may be adopted by event processing languages. For example, Esper [73] supports them. They propose various notions of context for an event (e.g., its temporal context or its spatial context). These notions of context, however, do not indicate external knowledge that may assign meaning to what the event represents. They are used as a way of partitioning events for further processing.

Other efforts aim at integrating CEP into the host language, making its constructs first class citizens [13]. For example, EventJava extends Java with language elements to support event-based programming [77]. EventJava supports unicasting and broadcasting of events and yields mechanisms to treat events as signals or as streams of data. It thus bridges the gap between the interpretation of events as pure signals and events solely as part of a stream of data. However, the constructs EventJava provides are intentionally very similar to CEP statements. Thus, they operate on the same level of abstraction as these statements.

Situational Awareness

Like event processing (cf. Section 2.2), situational awareness originates in armed flight, where it was important that a fighter was aware of the situation before his enemy was [100, 192]. It was thus mandatory for the pilot to know what was going on inside his system, which in this case is the plane. Today, situational awareness may also mean the inverse, like in sensor fusion [209]: situational awareness systems suggest a perspective shift in programming reactive applications. The idea is to describe a situation to the computer system and an action to take upon detection of this situation. In this regard, the goal of situational awareness is very similar to our policy-driven contextualization. Sometimes, situational awareness is also called situation monitoring.

For example, Georgakopoulos et al. describe a video awareness systems, in which users can specify *awareness specifications* which are then used to contextualize events [14, 98]. They use the awareness specification to combine primitive events into more complex ones, which they call *fusing*, to ultimately be aware of the specified situation. To deal with semantic heterogeneity, they use an event ontology, which captures semantically compatible event types. In their example application, the *Video Event Awareness Workbench* (VEAW), they provide a graphical user interface for defining awareness specifications. While their work is certainly among the most influential ones for this thesis (the scenario description in Section 1 is based on their scenario), their approach faces some drawbacks in light of today's distributed, dynamic and heterogeneous systems. Their system does not use a Message-oriented Middleware (MOM)

⁵ https://docs.tibco.com/pub/streambase_cep/7.4.1/doc/html/authoring/auth-operator-overview.html

for forwarding events: they hardwire information flows from specific sensors to specific processing components. Furthermore, the types of events, including the contextualized types, are hardcoded into the system and must thus be known a priori. Our approach on the other hand, provides runtime flexibility and uses modern messaging techniques for further decoupling.

Hasan et al. recognize the problem that primitive events (which they call sensor readings) do not contain enough information in themselves to be useful [109]. Their approach is similar to ours in that they let a user-defined situation guide which external information is added to the events. Their approach transforms sensor readings into RDF data items, which are then used to query an external linked data storage using spreading activation [58]. They use semantic similarity to the situation description and knowledge about the usefulness of previous enrichments to decide if an external data item should be added. Further factors are the expected impact on response time and the available memory. However, we believe that their approach faces some challenges: they use CEP queries (specifically Event Processing Language (EPL) statements) to describe the situations. Even though their approach allows for using complex events without defining them explicitly beforehand, they still face the problem of rule management and the technical details of EPL. Furthermore, their critical event processing path seems to be very long. Transforming events to RDF items, then matching these items with linked storage, performing elaborate filtering techniques like semantic similarity and using historic knowledge before letting a CEP engine process the enriched event probably impacts latency a lot. Sadly, they do not provide a performance analysis to showcase the feasibility of their approach. Another drawback is their limitation to enrichment of external, linked data. Thus, they cannot, for example, include a relational database. We believe that contextualizing events by taking other events into account as well as external sources is a more powerful approach.

In his dissertation, Micheal Olson describes a Situation Awareness Framework [146]. The framework is a cloud service that is able to integrate various event sources to make humans aware of certain situations. The situation awareness is realized by presenting the use with a dashboard-like interface, containing aggregated information from the event sources. New sensors, actuators and software modules for detection of interesting situations can be added to the cloud service. However, the software modules have to be developed by experts.

The early work "Situation monitoring for active databases" by Rosenthal et al. [167] is more on the foundation of event processing in general, than the description and evaluation of complex situations. The authors state that "a *situation* is an *event/condition* pair". Thus, while their approach allows for efficient processing of ECA rules, a developer has to write all of these rules.

Other situational awareness systems focus on one concrete situation, whose detection they aim to improve: gaining an overview during emergency situations like natural hazards [203, 204] or tracking of people and objects in multi-camera surveillance setups [202].

Simpler Development of Reactive and Cyber-physical Systems

Since reactive systems are complex, several researchers suggest methods to tackle this complexity. Ukuflow and Event Stream Processing Units make use of Business Process Model and Notation (BPMN). Ukuflow allows users to define workflows, which are then deployed and executed in a Wireless sensor and actor network (WSAN) [106]. Their target audience are also domain experts, aiming at providing them with a facility to program WSANs. Based on the definition of these workflows, their middleware defines node behavior and organizes the dataflow between nodes. Although targeted to WSANs, we believe the approach can be adopted to reactive systems in general. Similarly, Event Stream Processing

Units (SPUs) support developers and domain experts by introducing an abstraction for event stream processing [11]. Appel et al. suggest an extension to BPMN which makes event streams and event stream processing first class citizens of the notation [10]. Thus, domain experts and developers have a common abstraction they can use for communication and organizing systems. In addition, Appel et al. suggest a middleware to manage these components, calling them *Eventlets*, which are reusable event stream processing tasks [9]. Both systems are very promising and share the overall goal with us. However, we rely on declarative behavior descriptions, while Ukuflow and SPUs use a procedural approach. No approach is inherently superior over the other.

MobileFog abstracts from the distributed nature of applications for CPSs, which the authors call *Future Internet Applications*. While it clearly targets software developers it also acknowledges that the distributed nature of CPSs calls for additional support for developers. The authors thus introduce a programming model which abstracts from distribution. MobileFog is able to distribute applications written in this programming model on various components organized in a hierarchical network, based on the paradigm of *fog computing* [26].

Stehr et al. also acknowledge that programming CPSs is a complex task [193]. Similar to us, they suggest a declarative approach to define desired behavior. Their approach is based on logic programming, organizing CPSs into facts and goals. Sensors readings, for example, become facts, while queries for information or actions to take are represented as goals. Since facts are distributed throughout the system, they show how to match distributed facts against existing goals and describe how the distributed execution works. While their approach is very interesting, defining behavior requires a deep understanding of logic programming and writing a large number of rules. In contrast, we target domain experts and avoiding a collection of rules is one of our major requirements.

Talcott takes a more fundamental perspective and looks at possible foundations for the development of CPSs [197]. She concludes that event-based semantics provide a suitable foundation for CPSs and suggests two development approaches. One approach uses policy-based coordination between actors. In this approach, a policy is an instruction from a fixed set of communication instructions like *deliver* and *wait* [198]. The other approach also uses the same notion of policies for coordination and goals to drive behavior, similar to Stehr et al. Unfortunately, no details on the specification or implementation are given.

m3 is a middleware for reactive components [158]. The work on m3 mostly focuses on the coordination of reactive components. A key requirement is the ability to switch between different interaction modes at runtime. A developer of an m3-component specifies the context of interest, adaption rules, policy rules and coordination specifications. The coordination specifications are basically event-triggered message exchanges. For example, an account balance event which indicates that the balance is above 10000 emits a new event. The context is a key/value repository of certain environmental properties. Adaption rules are ECA rules, which in their action part call an adaption action. These actions are provided by the developer. The main goal of these adaption rules is to keep the detection of an adaption separate from its implementation. Finally, ECA-style policy rules are essentially obligations, prohibitions or permissions of roles for actions. While we think that the separation of ECA rules into predefined categories helps organizing the potentially large rule set, we still follow a more radical approach in providing a mechanism that abstracts from fine-grained rules.

Søberg et al. introduce CommonSens, a CEP system for automated home care [190]. CommonSens offers a mechanism to reuse existing CEP queries through personalization. For example, in automated homecare, queries for detecting the fall of a person or that medication was taken are written once and can then be adapted by a developer for the specific house. The approach of CommonSens is to define

higher-level situations like the fall of a person as CEP queries, with dependencies to certain kinds of sensors. A developer then only needs to personalize specific parts of that query (e.g., the `personID`). While the motivation of using simple, declarative statements to create reactive behavior, we see several shortcomings of CommonSens compared to our approach: CommonSens targets developers and does not make the hard part of writing CEP queries any easier. Furthermore, CommonSens's only target domain is automated home care, while we provide a flexible approach that can be used in various situations. We also do not limit the user to previously defined and compiled queries, but allow for a flexible way of describing behavior. Finally, CommonSens seems to pull data from its sensors ("and the system has to constantly pull the sensors" [190]), which is unsuitable for today's CPSs.

Obweiger et al. also observe that business users are challenged with the required technical skills to use EBSs. They advocate an approach in which business users use prepared building blocks to state their event-based business logic. Technical experts prepare these building blocks from low-level CEP rules. While we share their motivation, we argue that our approach takes a step further: business users do not just assemble their logic, which still requires profound knowledge about event-based systems, in our approach they just describe desired functionality.

The goal of behavioral programming is to create "reactive systems incrementally from their expected behaviors" [108]. A developer first programs basic behavior of the target application in self-contained modules. Behavior is then constructed by combining these individual modules. Harel et al. use a non-traditional form of event-based communication for the synchronization. At each synchronization point, each module defines three sets of event types: *requested* events, *waited-for* events and *blocked events*. Requested events are those the module would like to be triggered, while waited-for events are not requested to be triggered, but the module would still like to be informed when they happen. Finally, blocked events are those events that are forbidden to be triggered. Whenever all modules reach synchronization points, a middleware selects an event which is requested, but not blocked and informs the appropriate modules. Thus, small behavior-modules can be combined into more complex behavior. While we share the idea of expressing behavior on a high-level basis, behavioral programming still requires explicit coding and combining behaviors is still a complex task and even subject to deadlocks.

Policies as Declarative Statements

The idea of using policies as declarative statements is well established. Many other researchers also see the benefits of giving users an easy-to-use tool to specify complex system behavior. However, the systems employing policies to such means that we came across during our analysis of related work usually restrict policies to a fixed instruction set. The most prominent use case for policies is IT security.

SBUS is a communication middleware supporting publish/subscribe and request/reply interaction paradigms and uses policies to enforce restrictions on data flow [187]. Since multiple applications may use SBUS as their communication substrate, SBUS can enforce these policies across application boundaries. In addition, SBUS uses policies for coordinating multiple components to achieve higher-level goals. Originally, SBUS was developed as a information-flow management infrastructure for transport monitoring [186]. In SBUS, policies are triggered by events. The authors mention that the triggering event might be raised as a low-level message or as a result of more complex processing, including the integration of contextual knowledge. However, they provide no details on how these more complex *situations* can be detected, as this is not the focus of SBUS. Thus, we see the work on SBUS and this thesis as complementary work: while SBUS focuses on the actions taken by triggered policies, we provide a generic framework for detecting the triggering situations based on a high-level description.

Ponder uses policies for managing system behavior. [62, 70] Contrary to our approach, policies in Ponder "define choices in behavior in terms of the conditions under which predefined operations or actions can be invoked rather than changing the functionality of the actual operations themselves". Thus, Ponder uses policies to *select* between different behaviors, while we use policies to declaratively describe desired behavior. Ponder policies operate on domains, which are essentially groupings of objects according to some attribute. For example, grouping might happen based on the status of a person or based on the location (e.g., all persons in a certain room). We do not support such a grouping directly. However, it is always possible to model these groupings with a relationship (see Section 5.1). Ponder distinguishes three kinds of policies [61]: *access control* policies, *obligation* policies and *composite* policies. Access control policies define which subjects may, or may not do, which actions on which objects, or if certain information is filtered before further processing. Obligation policies are essentially ECA rules. They define actions upon the detection of a certain action of a subject on an object. Composite policies group other policies, for example along organizational structures. This helps managing policy sets. Since policies contain higher-level instructions, obligation policies together with composite policies share similar goals with our approach. However, Ponder policies still contain many technical details and have to rely on events already present in the system.

Rusello et al. use policies to describe interactions among application components [171]. They observed that in applications for Wireless sensor networks (WSNs), the actual application code (i.e. the code defining reactions to sensor readings) is mixed up with management code that deals with general concern like data distribution or resource utilization. They use policies to interconnect application components with middleware components and coordinate component activities in order to fulfill system-wide goals. The policies they use are essentially stateful ECA rules. They allow users to specify conditions on the *direction* of an event. Thus, it makes a difference if an event is sent from a broker to another broker or from a broker to an application component. This is different from the usual use of ECA rules inside application-level consumers, but serves their goal of being able to coordinate various components. However, their policies share the rather low abstraction of common ECA rules. Thus, we operate on a might higher-level abstraction and use policies for higher-level situation descriptions.

Another use for policies in WSNs is discussed by Matthys et al., suggesting a policy-driven approach for reconfiguration [135]. They argue that dynamic application customization by adding or removing only individual software components is very beneficial for WSNs, because replacing an entire node image is expensive. This of course raises the question of management and how a varying set of components interacts. In their suggested middleware CaPI, components define application behavior and are loosely coupled, using an event-based communication. Policies can then be used to intercept events, allowing for blocking them or rewiring component interactions entirely. In this way, CaPI policies share functionality with aspect-oriented programming [121]. Policies follow an ECA pattern and contain concrete, low-level instructions in their action-part. Thus, these policies are neither declarative nor understandable for domain experts.

Belokosztolszki et al. describe a role-based access control extension for publish/subscribe systems, which makes use of policies [16]. Event types have dedicated owners, who can define which other roles have access to that type. Thus, clients may be limited in their advertisements or subscriptions, if the roles they have do not have sufficient rights. Policies are used to capture access rights. Policies are limited to these access rights specifications and cannot describe any higher level behavior beyond that of access control.

The Alarm Correlation Engine (ACE) aims at improving hardware network management by providing automatic diagnostics and thus reducing reaction times [213]. The target users are field technicians, who are no computer experts. Like our approach, the ACE employs a domain specific language which

lets users focus on what they want to achieve and not how this is done. Since ACE's goal is to correlate individual alarm messages that in reality correspond to the same fault, users specify correlation patterns. These patterns can be seen as declarative policies and are stored in a correlation database. Patterns consist of a firing condition and an actions part. Firing conditions consist of the alarm types, their number and frequency as well as canceling conditions. Thus, the ACE also aims at giving domain experts a tool for specifying policies and builds policies from a situation description and an actions part. However, ACE's policies are highly tailored to network management and cannot be used for other domains. In addition, it does not support any external knowledge sources beyond those that produce network alarms.

Other Related Work

Handling multiple, potentially heterogeneous data sources with the help of an ontology has been explored in the field of query formulation [133, 44]. Users are supported with suggestions to the queries they are trying to construct. Among other hints, query formulation systems suggest vocabulary based on the underlying ontology.

Policies can be viewed as a conceptual grouping mechanism of CEP queries. For example, the policies given in Chapter 5 can be seen as a collection (group) of the Event Query Language (EQL) statements. In that regard, our approach is similar to constraint grouping techniques in Database Management Systems (DBMSs) [38, 36]. Constraint grouping aims at grouping database constraints into meaningful units, which can be plugged in and out, without worrying about other dependencies. However, constraint grouping and query formulation have been designed for pull-based interactions, while we specifically target push-based interaction.

7 Conclusion and Future Work

*The Web as I envisaged it, we have not seen it yet.
The future is still so much bigger than the past.*

–Tim Berners-Lee

Modern developments in information technology constantly challenge established concepts and methods. Trends like cloud computing and the Internet of Things (IoT) have changed how software systems are developed and how they work. Applications have progressed from a monolithic binary that runs on a single machine to vastly distributed interactions of small, independent software components. This transformation allows for much more powerful and pervasive uses of modern technology, but comes with the cost of significantly increased complexity. This makes middleware ever more indispensable.

Event-driven Architectures (EDAs) promise to provide an abstraction that allows for handling the vast amount of data that Cyber-physical Systems (CPSs) will produce. However, event-enabled middleware is often hard to use for computer scientists and impossible to use for domain experts. Just like users manipulate the physical world around them, they need to manipulate the CPSs at their disposal. Despite that, most research effort in Event-based Systems (EBSs) is put into improving performance. Thus, in this thesis, we took an approach that simplifies engineering and use of EBSs.

The main problem we identified is that the first class citizens of EBSs – the events – do not carry enough information to be properly understood. We identified two main causes for this missing information: a very low abstraction and the distributed and dynamic nature of these systems.

We analyzed the importance of context to properly understand events. As a first step, we looked at existing context-aware systems and how they relate to EBS. We found that we need to differentiate between context about data and context about events, which reflects the two causes for missing information. Context about data defines the semantics of the data and helps with the interpretation. Context about the data in an event thus helps to avoid the drawbacks of anonymity in EBSs without sacrificing its benefits. Context about the event itself is information that is not carried by the event, but is still of interest to the application. By adding context about the event to the event itself, we can raise the abstraction level of events.

We examined many established context-aware systems and surveys about these kinds of systems to establish structural requirements for any middleware that aims at using context for EBSs. We distinguished between requirements for context about data and requirements for context about events. Furthermore, we compiled a general architecture of context-aware systems based on the research papers and used this to devise an architecture recommendation for using context in EBSs.

In traditional request/reply communication, the communication partner is usually known, which is the only context information necessary for correct data interpretation. In this thesis, we presented a framework for publish/subscribe systems that makes assumptions about the semantics of the data explicit. We rely on automatic transformations by the broker to mediate between publishers and subscribers of information. Our framework defines interpretation contexts which are assembled from four components: local types, type mappings, transformation rules and transformation functions. This separation allows

for fine-grained adaptation and reuse of interpretation contexts. Furthermore, we have clear and intuitive priorities among possibly conflicting rules, which both increase flexibility within the framework while at the same time making definition of interpretation contexts easier. To aid developers even more, we integrated the framework into Java using annotations. While our annotations do not provide the complete expressivity of our rules and type mappings, they support a large, very common subset of use cases.

We call our implementation of the transformation framework ACTrESS. ACTrESS builds upon industry-strength publish/subscribe middleware and uses a plugin to intercept messages. The intercepted messages are transformed according to the given interpretation context for the client. Thus, once setup, clients do not need to worry about data semantics and can simply assume data to follow their assumptions. The evaluation of our approach shows that it simplifies the development of EBSs by reducing the effort necessary to transform between different interpretations of data. Furthermore, the performance of our transformations is high. Since we generate and compile a dedicated transformer after analyzing a client's interpretation context, the actual transformation is quick and lightweight. In fact, performance in a content-based publish/subscribe system is indistinguishable between using and not using ACTrESS. We thus created a provably type-safe transformation mechanism with a formal foundation, that can be integrated into industry-strength Message-oriented Middleware (MOM), is very fast while still supporting the full dynamism of EBSs, and solves the data interpretation problem. Our approach fulfills all the identified requirements.

Since primitive events usually have a very low level of abstraction and do not yield enough information on their own to be useful, they are combined with external information or composed with other events. This leads to complex events of higher abstraction which eventually are able to represent a meaningful happening. The problem with this approach is that the rules which define such composition have invisible dependencies, are hard to understand and harder to write. This prevents domain experts from using EBSs in a meaningful way; they have to rely on computer experts. We realized that humans usually think in terms of *situations* when talking about reactive behavior. Thus, we designed an approach which allows users to define a situation in terms of the participating concepts and their relation to each other. We call these descriptions policies, which include an action to execute, once the defined situation is detected. Our middleware uses policies to decide which external information to add to events and how to compose events into higher-level events. To do that, a computer expert has to setup the middleware by providing a domain model. After that, however, domain experts can specify policies in a declarative manner, enabling them to focus on what they want to achieve rather than how to express it.

Our middleware generates code for many, small enrichment components which we call Event Enrichment Components (EECs). Each EEC is decoupled from the others which allows for easy distribution. Since EECs are managed by our middleware, this distribution comes at no additional effort to users, but enables easy load management. EECs communicate by sending messages, which makes long processing chains of EECs incur a significant impact on latency. To compensate for that, our middleware arranges the policy in a way that guarantees a maximum of five EEC-hops from primitive event to situation detection. Our code generation respects identities of concepts and handles NOT statements without injecting artificial NOT-events into the system. The evaluation of our approach shows promising results. We evaluated our policy language using the Cognitive Dimensions framework, comparing it with Esper EPL and Java. We found that our policy language is indeed easier to use and understand as it aligns well with the mental model of humans. A user study we conducted supports this analysis. Compared to Java and EPL, participants could better understand our policy language and found it easier to understand concrete situations. Compared with the highly optimized Esper, we incur more CPU utilization, but are much better at parallelization.

In summary, we simplify the development and use of EBSs in two ways: by making assumptions about data semantics explicit and capturing them in interpretation contexts, we automatically transform between different assumptions. Furthermore, we designed a contextualization model, which uses declarative situation descriptions to drive and guide the contextualization of events. These situation descriptions significantly raise the level of abstraction at which reactive behavior can be specified. We thus enable CPSs to step beyond a computer scientist's domain and make the Internet of Things accessible for everyone.

7 Future Work

Future work related to this thesis can be divided into considerations about our transformation framework ACTrESS and policy-driven contextualization of events.

7.1.1 ACTrESS

Optimal Transformation Placement in Distributed Publish/Subscribe

Currently, we employ a greedy transformation approach: messages are transformed by the first broker they reach. This approach works very well in most cases, but we identified topologies, in which such an approach leads to unnecessary work. For example, a producer publishing an event that two consumers will receive, one of which shares the interpretation context with the producer. The transformation back to the original interpretation context is unnecessary in this case. We want to develop strategies of where to transform. However, this is not a straightforward decision. Even when we assume global knowledge, there are still a number of questions to be solved. How do we do the matching if we try to avoid unnecessary work? Do brokers keep a transformed and original version of the subscription? Does the additional matching effort outweigh the savings achieved? Do we send the transformed message along with the original message? What is the cost of increased message size? If we add dynamic adaptation to the current load into the equation, things become even more complicated. Nonetheless, we believe that it is worthwhile to carefully analyze this problem and suggest strategies which lead to good or even optimal transformation placement.

Integration with EAI

Enterprise Application Integration (EAI) defines various concepts and architectures for integrating different software components into a software landscape. It also supports messaging between endpoints. However, possible transformations always happen on static routes between known endpoints. Even though an endpoint may be a JMS topic and thus the actual subscribers can be anonymous to the system, all subscribers on that topic have to share the same data interpretation. EAI in its current form simply does not support full anonymity and flexibility in transformations like ACTrESS does. However, it is an established concept and known in industry. We thus believe it may be worthwhile to see how the concepts of ACTrESS can be integrated into the EAI framework and how the framework needs to be adjusted.

7.1.2 Policy-driven Contextualization

Event Reuse

Currently, policies execute in isolation. This is helpful to ensure correct execution and rule out interference. However, in larger systems, there may be parts of policies which state the same thing. For example,

employee is inside room may be a part in multiple policies. In the current implementation, we would construct an EEC for each policy, even though they are identical. Not only is this unnecessary generation work, but more importantly, more computation work.

A straightforward idea is thus to reuse EECs among policies. This involves keeping track of which EECs have already been generated and which policies are using which EEC. This most likely needs a dependency graph between policies and EECs. Furthermore, this idea requires thinking about the communication model. When using JMS, one has to ensure that no other parts of processing policy A leak into the processing of policy B. When using sockets, EECs will have to support one-to-many communication. This may cause sockets to be the less favorable choice in such a scenario. As we can see, exploring this idea creates many questions, which are worth exploring.

Policy Conflict Detection

As already stated in Section 5.3.2 policies may conflict. We believe that an elaborate conflict detection would help multi-user environments. Since this is a one-time effort before the actual policy processing, elaborate, computationally expensive mechanisms can be used. Policies can conflict in multiple ways: a) Policies may form cycles. If the action of a policy causes another policy to trigger, whose action in turn triggers the first policy, the policies have formed a cycle. b) Policies may try to execute contrary actions. For example, one policy may open a window while another one closes it. c) Policies may try to execute actions whose *effects* conflict. There is some work on detecting cycles in Event-Condition-Action (ECA) rules, which can be adopted to policies [185]. A first step in detecting a) and c) consists in constructing a mechanisms which is able to analyze possible states of the situations policies describe, if two policies can be simultaneously triggered, then an analysis of their actions is required, to see if these actions conflict. For c this requires a way of capturing the effect of an action. This is a hard problem, because actions may have secondary effects beyond the obvious. For example, opening a window surely has the effect that the window is open. Depending on the weather outside, however, a secondary effect may be the cooling or warming of the room. As the example shows, trying to capture the effect of an action also requires contextual knowledge. Other computer science disciplines like Natural Language Processing or Agent-based Programming are most likely helpful for tackling this problem.

Window Sizes

Window sizes define how long to wait for events in case of a correlation and how long to wait in case of NOT-relationships. In our security example, a smaller window size means that an employee has less time to get close to a guest in a restricted room before the alarm is sounded. This might be desired behavior or lead to false positives. Likewise, a longer window may prevent false positives but cause the alarm to be sounded too late.

So far, our approach relies on a configurable, but fixed window size. This is fine as long as the computer expert who sets up the system knows a sensible window size for all policies that will be stated in the system. However, we believe that our system would benefit from allowing for more flexibility. Making the window size more flexible raises a few questions though. Do we want to support flexibility on a per policy or even a per operator basis? Surely, more flexibility raises the expressive power of our language, but since usability was one of our main requirements, we are not certain how flexibility on a per operator basis would impact understandability of our language. Another question is how the window size is determined. An explicit statement in the policy is a straightforward idea. We also envision approaches which would use semantic knowledge to infer a window size from the policy itself.

Distributing Further: Eventlets

In addition to distributing EECs, we can split the work of EECs even further, by using the Eventlet paradigm. Eventlets were mainly developed by Stefan Appel for his dissertation [8]. We collaborated with Mr. Appel on this topic and produced a number of publications together ([9, 10, 11]). We want to introduce Eventlets here and then illustrate how using this approach helps with further distributing the work of EECs.

Observing the architectural support when developing applications, we noticed that the "pull-world" enjoys the support of many paradigms. For example Service-oriented Architectures (SOAs) help with decoupling and reusing application components. There is little such support in the push-based world. Eventlets alleviate this discrepancy by encapsulating event-driven tasks in a similar manner as (SOA-)services encapsulate business processes. Eventlets are different to event-driven SOA, as in the latter events are occasional triggers for services, whereas in the former, *event streams* are continuously processed according to a task definition.

Figure 7.1a illustrates the concept of event-driven tasks and Eventlets. A task is triggered by events and causes certain actions to be applied to entities. Tasks are defined in a generic way and a concrete task instance is automatically instantiated for each individual entity. The generically defined actions are automatically applied to each identified instance. Figure 7.1b shows a concrete example of a leaf-EEC framed as an Eventlet task. The abstract task is to check the status of a person and, if conditions are met, produce an enriched event (cf. Section 5.3.3). An instance of this task is created for each person.

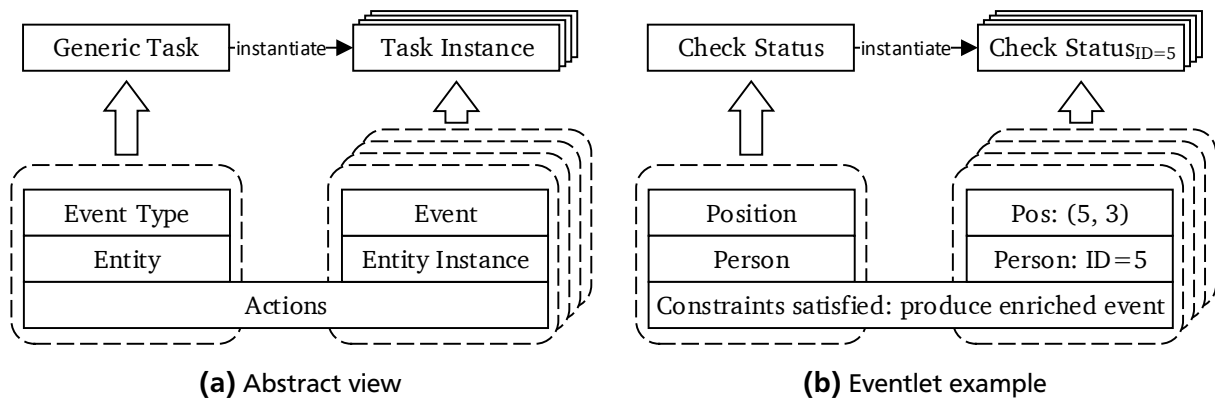


Figure 7.1.: Illustration of the generic task model of Eventlets [8]

Eventlets have a defined lifecycle managed by a middleware. Developers can register new Eventlets at the middleware by writing Eventlet *prototypes*. An Eventlet prototype encodes the generic event-driven task in a structured way which can be understood by the middleware.

By using Eventlets, we can thus not only distribute EECs, but even distribute the processing of a single EEC. Since it is impossible to know all possible ID values a priori, Eventlet instances are created on demand. This however, proves challenging for EECs requiring two different kinds of events, e.g., a correlation-EEC. It is easy to define the instantiation expression as a compound of the two contributing event's identifying attributes. But events do not arrive at the same time and thus checking the compound is not straightforward. Instead, the Eventlet middleware has to be adapted, so it waits for the second event to happen and then instantiates the Eventlet, using the compound of the contributing events as the instantiation value. Since waiting requires a window size, the Eventlet middleware must be able to communicate with our own middleware.

Event Authenticity

It is always desirable to create secure systems. In this thesis, we did not talk about IT security aspects like authentication, privacy, security or authenticity. Incorporating all these aspects was simply out of scope of this thesis and existing concepts of publish/subscribe systems can be readily applied. However, message authenticity is hampered by our approach, because we rely on changing messages, either through transformation or enrichment. An easy approach would be to give our components a valid signing certificate with which messages can be signed. The disadvantage of this approach is that every event would then seem to originate from the broker, without being able to hold the original sender accountable.

In previous work, we redesigned a method called sanitizable signatures [33]. The idea of sanitizable signatures is that the original signer allows the signed document to be modified in certain ways at pre-defined places. These modifications include blackening, changing values or any other, arbitrary changes. The motivating example is patient statistics that hospitals give to governmental institutions. Doctors sign the documents to ensure their authenticity, but allow hospital administration staff to replace identifying attributes (like the patient's name) with pseudonyms. We can use this for our approach to allow the broker/the policy engine to modify certain parts of the event (e.g., adding attributes), without losing authenticity towards the original senders.

The paper focuses on the formal treatment of sanitizable signatures, providing proof that they obey the established security requirements unforgeability (signatures cannot be forged), immutability (only admissible parts may be changed), privacy (original parts cannot be reestablished), transparency (original signer and sanitizer cannot be distinguished) and accountability (in case of a dispute, a *judge* can decide the origin of a valid signature). For the purposes of our approach, privacy is not important and while transparency is supported by the scheme it can easily be removed by simply adding the signer's name to the event. This does not prevent accountability, as the judge is given the secret signing keys to make their decision. Due to the focus on the formal treatment, we did not have a working implementation which we could incorporate, but we certainly see the benefit of providing support for sanitizable event signatures in the future.

Bibliography

- [1] Gregory D Abowd, Anind K Dey, Peter J Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a Better Understanding of Context and Context-awareness. In *Handheld and ubiquitous computing*, pages 304–307. Springer, 1999.
- [2] Raman Adaikkalavan and Sharma Chakravarthy. Seamless Event and Data Stream Processing: Reconciling Windows and Consumption Modes. In *Database Systems for Advanced Applications*, pages 341–356. Springer, 2011.
- [3] J.G. Adair, D.J. Coyle Jr, R.J. Grafe, B.G. Lindsay, R.A. Reinsch, R.P. Resch, P.G. Selinger, and M.R. Zimowski. Heterogenous Database Communication System in Which Communicating Systems Identify Themselves and Convert any Requests/Responses Into Their own Data Format, 1995.
- [4] Asaf Adi and Opher Etzion. Amit – The Situation Manager. *The VLDB Journal – The International Journal on Very Large Data Bases*, 13(2):177–203, 2004.
- [5] Eric Allen, Justin Hilburn, Scott Kilpatrick, Victor Luchangco, Sukyoung Ryu, David Chase, and Guy Steele. Type Checking Modular Multiple Dispatch With Parametric Polymorphism and Multiple Inheritance. In *ACM SIGPLAN Notices*, volume 46, pages 973–992. ACM, 2011.
- [6] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. Web Services. In *Web Services, Data-Centric Systems and Applications*, pages 123–149. Springer Berlin Heidelberg, 2004.
- [7] Jose Antollini, Mario Antollini, Pablo Guerrero, and Mariano Cilia. Extending REBECA to Support Concept-Based Addressing. In *Proceedings of the Argentinean Symposium on Information Systems*, September 2004.
- [8] Stefan Appel. *Integration of Event Processing with Service-oriented Architectures and Business Processes*. PhD thesis, Technische Universität Darmstadt, Darmstadt, Germany, May 2014.
- [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)*, Taipei, Taiwan, 2012. Best Paper Award.
- [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)*, Beijing, China, 2013.
- [11] Stefan Appel, Pascal Kleber, Sebastian Frischbier, Tobias Freudenreich, and Alejandro Buchmann. Modeling and Execution of Event Stream Processing in Business Processes. *Information Systems*, 2014.
- [12] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A Survey. *Computer Networks*, 54(15):2787–2805, 2010.
- [13] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A Survey on Reactive Programming. *ACM Computing Surveys*, 45(4):1–34, August 2013.

-
- [14] Donald Baker, Dimitrios Georgakopoulos, Marian Nodine, and Andrzej Cichocki. From Events to Awareness. In *Services Computing Workshops, 2006. SCW'06. IEEE*, pages 21–30. IEEE, 2006.
- [15] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. A Survey on Context-aware Systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, 2007.
- [16] András Belokosztolszki, David M Eyers, Peter R Pietzuch, Jean Bacon, and Ken Moody. Role-based Access Control for Publish/Subscribe Middleware Architectures. In *Proceedings of the 2nd International Workshop on Distributed Event-based Systems*, pages 1–8. ACM, 2003.
- [17] Amel Bennaceur, Gordon Blair, Franck Chauvel, Huang Gang, Nikolaos Georgantas, Paul Grace, Falk Howar, Paola Inverardi, Valrie Issarny, Massimo Paolucci, Animesh Pathak, Romina Spalazzese, Bernhard Steffen, and Bertrand Souville. Towards an Architecture for Runtime Interoperability. In *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6416, pages 206–220. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
- [18] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The Semantic Web. *Scientific American*, 284(5):28–37, 2001.
- [19] Claudio Bettini, Oliver Brdiczka, Karen Henriksen, Jadwiga Indulska, Daniela Nicklas, Anand Ranganathan, and Daniele Riboni. A Survey of Context Modelling and Reasoning Techniques. *Pervasive and Mobile Computing*, 6(2):161–180, 2010.
- [20] Claudio Bettini and Daniele Riboni. Profile Aggregation and Policy Evaluation for Adaptive Internet Services. In *Mobile and Ubiquitous Systems: Networking and Services, 2004. MOBIQUITOUS 2004. The First Annual International Conference on*, pages 290–298. IEEE, 2004.
- [21] Andrew D Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.
- [22] Gordon Blair, Geoff Coulson, and Paul Grace. Research Directions in Reflective Middleware: the Lancaster Experience. In *RM'04*, 2004.
- [23] Gordon S. Blair, Amel Bennaceur, Nikolaos Georgantas, Paul Grace, Valérie Issarny, Vatsala Nundloll, and Massimo Paolucci. The Role of Ontologies in Emergent Middleware: Supporting Interoperability in Complex Distributed Systems. In *Middleware 2011*, volume 7049, pages 410–430. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
- [24] C. Bolchini, C.A. Curino, E. Quintarelli, F.A. Schreiber, and L. Tanca. Context Information for Knowledge Reshaping. *International Journal of Web Engineering and Technology*, 5(1):88, 2009.
- [25] Cristiana Bolchini, Carlo A. Curino, Elisa Quintarelli, Fabio A. Schreiber, and Letizia Tanca. A Data-oriented Survey of Context Models. *ACM SIGMOD Record*, 36(4):19, December 2007.
- [26] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog Computing and Its Role in the Internet of Things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.
- [27] Christof Bornhovd. Semantic Metadata for the Integration of Web-based Data for Electronic Commerce. In *WECWIS'99*, 1999.
- [28] John R Boyd. *A Discourse on Winning and Losing*. 1987.

-
- [29] Karin Koogan Breitman, Alu  zio Haendchen Filho, Edward Hermann Haeusler, and Arndt Staa. Using Ontologies to Formalize Services Specifications in Multi-agent Systems. In *Formal Approaches to Agent-Based Systems*, volume 3228, pages 92–110. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
- [30] S. Bressan, C. H. Goh, K. Fynn, M. Jakobisiak, K. Hussein, H. Kon, T. Lee, S. Madnick, T. Pena, J. Qu, A. Shum, and M. Siegel. The Context Interchange Mediator Prototype. In *SIGMOD’97*, Tucson, Arizona, United States, 1997.
- [31] Peter J Brown. The Stick-e Document: A Framework for Creating Context-aware Applications. *Electronic Publishing -Chichester-*, 8:259–272, 1995.
- [32] P.J. Brown, J.D. Bovey, and Xian Chen. Context-aware Applications: From the Laboratory to the Marketplace. *Personal Communications, IEEE*, 4(5):58–64, Oct 1997.
- [33] Christina Brzuska, Marc Fischlin, Tobias Freudenreich, Anja Lehmann, Marcus Page, Jakob Schelbert, Dominique Schr  der, and Florian Volk. Security of Sanitizable Signatures Revisited. In *Public Key Cryptography–PKC 2009*, pages 317–336. Springer, 2009.
- [34] Martin B  chi and Wolfgang Weck. Compound Types for Java. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA ’98, pages 362–373, New York, NY, USA, 1998. ACM.
- [35] A. Buchmann, H. Chr. Pfohl, S. Appel, T. Freudenreich, S. Frischbier, I. Petrov, and C. Zuber. Event-Driven Services: Integrating Production, Logistics and Transportation. In *Service-Oriented Computing*, volume 6568, pages 237–241. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
- [36] A. P. Buchmann, R. S. Carrera, and M. A. Vazquez-Galindo. A Generalized Constraint and Exception Handler for an Object-oriented CAD-DBMS. In *Proceedings on the 1986 international workshop on Object-oriented database systems*, OODS ’86, pages 38–49, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.
- [37] Alejandro Buchmann, Stefan Appel, Tobias Freudenreich, Sebastian Frischbier, and Pablo E Guerrero. From Calls to Events: Architecting Future BPM Systems. In *Business Process Management*, pages 17–32. Springer, 2012.
- [38] Alejandro P Buchmann and Concepcion Perez de Celis. An Architecture and Data Model for CAD Databases. In *Proceedings of the 11th international conference on Very Large Data Bases - Volume 11*, VLDB ’85, pages 105–114, Stockholm, Sweden, 1985. VLDB Endowment.
- [39] Ioana Burcea, Milenko Petrovic, and Hans-Arno Jacobsen. I Know What You Mean: Semantic Issues in Internet-scale Publish/Subscribe Systems. In *SWDB*, pages 51–62, 2003.
- [40] L. Cardelli. Structural Subtyping and the Notion of Power Type. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’88, pages 70–79, New York, NY, USA, 1988. ACM.
- [41] Antonio Carzaniga, David S Rosenblum, and Alexander L Wolf. Achieving Scalability and Expressiveness in an Internet-scale Event Notification Service. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 219–227. ACM, 2000.
- [42] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and Evaluation of a Wide-area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

-
- [43] Antonio Carzaniga and Alexander L Wolf. Forwarding in a Content-based Network. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 163–174. ACM, 2003.
- [44] Tiziana Catarci, Tania Mascio, Enrico Franconi, Giuseppe Santucci, and Sergio Tessaris. An Ontology Based Visual Tool for Query Formulation Support. In *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, volume 2889, pages 32–33. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.
- [45] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *Proceedings of the International Conference on Very Large Data Bases*, pages 606–606, 1994.
- [46] K. Chandy. A Web That Senses and Responds. In *From Active Data Management to Event-Based Systems and More*, volume 6462 of *Lecture Notes in Computer Science*, pages 78–84. Springer Berlin / Heidelberg, 2010.
- [47] K Chandy and W Schulte. *Event Processing: Designing IT Systems for Agile Companies*. McGraw-Hill, Inc., 2009.
- [48] K Mani Chandy. Event-driven Applications: Costs, Benefits and Design Approaches. In *Gartner Application Integration and Web Services Summit*, San Diego, USA, 2006.
- [49] Harry Chen, Tim Finin, and Anupam Joshi. An Ontology for Context-aware Pervasive Computing Environments. *The Knowledge Engineering Review*, 18(03):197–207, 2003.
- [50] Peter Pin-Shan Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.
- [51] Chin-Wan Chung. DATAPLEX: An Access to Heterogeneous Distributed Databases. *Communications of the ACM*, 33(1):70–80, January 1990.
- [52] Mariano Cilia, Mario Antollini, Christof Bornhövd, and Alejandro Buchmann. Dealing with Heterogeneous Data in Pub/Sub Systems: The Concept-Based Approach. In *International Workshop on Distributed Event-Based Systems (DEBS04)*, May 2004.
- [53] Mariano Cilia, Christof Bornhövd, and Alejandro Buchmann. CREAM: An Infrastructure for Distributed, Heterogeneous Event-Based Applications. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 482–502. Springer Berlin / Heidelberg, 2003.
- [54] James Clark. XSL Transformations (XSLT). *World Wide Web Consortium (W3C)*, November 1999. <http://www.w3.org/TR/xslt>.
- [55] William F Clocksin, Christopher S Mellish, and WF Clocksin. *Programming in PROLOG*, volume 5. Springer, 1987.
- [56] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your Mediators Need Data Conversion! *SIGMOD Rec.*, 27(2):177–188, June 1998.
- [57] Edgar F Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.
- [58] Allan M Collins and Elizabeth F Loftus. A Spreading-activation Theory of Semantic Processing. *Psychological review*, 82(6):407, 1975.

-
- [59] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS. *IEEE Trans. Software Eng.*, 27(9):827–850, 2001.
- [60] Gianpaolo Cugola and Alessandro Margara. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.
- [61] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder Policy Specification Language. In *Policies for Distributed Systems and Networks*, pages 18–38. Springer, 2001.
- [62] Nicodemos Damianou, Naranker Dulay, Emil Lupu, Morris Sloman, and Toshio Tonouchi. Tools for Domain-based Policy Management of Distributed Systems. In *Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP*, pages 203–217. IEEE, 2002.
- [63] Umeshwar Dayal, Barbara Blaustein, Alex Buchmann, Upen Chakravarthy, Meichun Hsu, R Ledin, Dennis McCarthy, Arnon Rosenthal, Sunil Sarin, Michael J. Carey, et al. The HiPAC Project: Combining Active Databases and Timing Constraints. *ACM Sigmod Record*, 17(1):51–70, 1988.
- [64] Umeshwar Dayal, Alejandro P Buchmann, and Dennis R McCarthy. Rules are Objects Too: A Knowledge Model for an Active, Object-oriented Database System. In *Advances in Object-Oriented Database Systems*, pages 129–143. Springer, 1988.
- [65] Umeshwar Dayal and Hai-Yann Hwang. View Definition and Generalization for Database Integration in a Multidatabase System. *IEEE Transactions on Software Engineering*, SE-10(6):628–645, November 1984.
- [66] Nigel Deakin. Java Message Service - Version 2.0. Technical report, Oracle, March 2013.
- [67] Alan J Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, Walker M White, et al. Cayuga: A General Purpose Event Monitoring System. In *CIDR*, volume 7, pages 412–422, 2007.
- [68] Patricia Derler, Edward A Lee, and A Sangiovanni Vincentelli. Modeling Cyber-physical Systems. *Proceedings of the IEEE*, 100(1):13–28, 2012.
- [69] Anind K Dey. Context-aware Computing: The CyberDesk Project. In *Proceedings of the AAAI 1998 Spring Symposium on Intelligent Environments*, pages 51–54, 1998.
- [70] Naranker Dulay, Emil Lupu, Morris Sloman, and Nicodemos Damianou. A Policy Deployment Model for the Ponder Language. In *Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on*, pages 529–543. IEEE, 2001.
- [71] Michael Eckert, François Bry, Simon Brodt, Olga Poppe, and Steffen Hausmann. A CEP Babelfish: Languages for Complex Event Processing and Querying Surveyed. In *Reasoning in Event-Based Distributed Systems*, volume 347 of *Studies in Computational Intelligence*, pages 47–70. Springer Berlin Heidelberg, 2011.
- [72] Mica R Endsley. Designing for Situation Awareness in Complex Systems. In *Proceedings of the Second International Workshop on symbiosis of humans, artifacts and environment*, 2001.
- [73] EsperTech. Esper. <http://esper.codehaus.org/>.
- [74] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., 2010.
- [75] Patrick Eugster, Tobias Freudenreich, Sebastian Frischbier, Stefan Appel, and Alejandro Buchmann. Sound Transformations for Message Passing Systems. Technical report, TU Darmstadt, <http://www.dvs.tu-darmstadt.de/publications/pdf/transsem.pdf>, 2012.

-
- [76] Patrick Eugster and Rachid Guerraoui. Content-Based Publish/Subscribe with Structural Reflection. In *COOTS*, volume 1, 2001.
- [77] Patrick Eugster and KR Jayaram. EventJava: An Extension of Java for Event Correlation. In *ECOOP 2009—Object-Oriented Programming*, pages 570–594. Springer, 2009.
- [78] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.
- [79] David Eysers, Tobias Freudenreich, Alessandro Margara, Sebastian Frischbier, Peter Pietzuch, and Patrick Eugster. Living in the Present: On-the-fly Information Processing in Scalable Web Architectures. In *Proceedings of the 2nd International Workshop on Cloud Computing Platforms*, page 6. ACM, 2012.
- [80] Eli Fidler, Hans-Arno Jacobsen, Guoli Li, and Serge Mankovski. The PADRES Distributed Publish/-Subscribe System. In *In 8th International Conference on Feature Interactions in Telecommunications and Software Systems*, pages 12–30, 2005.
- [81] Ludger Fiege, Gero Mühl, and Felix C Gärtner. Modular Event-based Systems. *The Knowledge Engineering Review*, 17(04):359–388, 2002.
- [82] Michael Fitzgerald. An Internet for Manufacturing. Technical report, MIT Technology Review, 2013. <http://www.technologyreview.com/news/509331/an-internet-for-manufacturing/>.
- [83] Charles Forgy and John P McDermott. OPS, A Domain-Independent Production System Language. In *IJCAI*, volume 5, pages 933–939, 1977.
- [84] Charles L Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial intelligence*, 19(1):17–37, 1982.
- [85] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem. *ACM Transactions on Programming Languages and Systems*, 29(3), May 2007.
- [86] David Franklin and Joshua Flaschbart. All Gadget and no Representation Makes Jack a Dull Environment. In *Proceedings of the AAAI 1998 Spring Symposium on Intelligent Environments*, pages 155–160, 1998.
- [87] Tobias Freudenreich, Stefan Appel, Sebastian Frischbier, and Alejandro P Buchmann. ACTrESS - Automatic Context Transformation in Event-Based Software Systems. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 179–190. ACM, 2012.
- [88] Tobias Freudenreich, Stefan Appel, Sebastian Frischbier, and Alejandro P Buchmann. Using Policies for Handling Complexity of Event-Driven Architectures. In *Software Architecture*, pages 114–129. Springer, 2014.
- [89] Tobias Freudenreich, Patrick Eugster, Sebastian Frischbier, Stefan Appel, and Alejandro Buchmann. Implementing Federated Object Systems. In *ECOOP 2013—Object-Oriented Programming*, pages 230–254. Springer, 2013.
- [90] Tobias Freudenreich, Pedro Furtado, Christian Koncilia, Maik Thiele, Florian Waas, and Robert Wrembel. An On-Demand ELT Architecture for Real-Time BI. In *Enabling Real-Time Business Intelligence*, pages 50–59. Springer, 2013.
- [91] Ernest Friedman-Hill. *JESS in Action*. Manning Greenwich, CT, 2003.

-
- [92] Sebastian Frischbier, Alessandro Margara, Tobias Freudenreich, Patrick Eugster, David Eysers, and Peter Pietzuch. ASIA: Application-specific Integrated Aggregation for Publish/Subscribe Middleware. In *Proceedings of the Posters and Demo Track*, page 6. ACM, 2012.
- [93] Sebastian Frischbier, Alessandro Margara, Tobias Freudenreich, Patrick Eugster, David Eysers, and Peter Pietzuch. Aggregation for Implicit Invocations. In *Proceedings of the 12th annual international conference on Aspect-oriented software development*, pages 109–120. ACM, 2013.
- [94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Pearson Education, 1994.
- [95] T. Gannon, S. Madnick, A. Moulton, M. Siegel, M. Sabbouh, and Hongwei Zhu. Framework for the Analysis of the Adaptability, Extensibility, and Scalability of Semantic Information Integration and the Context Mediation Approach. In *HICSS'09*, 2009.
- [96] David Garlan, Daniel P Siewiorek, Asim Smailagic, and Peter Steenkiste. Project Aura: Toward Distraction-free Pervasive Computing. *Pervasive Computing, IEEE*, 1(2):22–31, 2002.
- [97] Stella Gatziau and Klaus R Dittrich. *Events in an Active Object-oriented Database System*. Springer, 1994.
- [98] Dimitrios Georgakopoulos, Donald Baker, Marian Nodine, and Andrzej Cichoki. Event-driven Video Awareness Providing Physical Security. *World Wide Web*, 10(1):85–109, March 2007.
- [99] Joseph Gil and Itay Maman. Whiteoak: Introducing Structural Typing Into Java. In *OOPSLA*, 2008.
- [100] Richard D Gilson. Special Issue Preface. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 37(1):3–4, 1995.
- [101] Paul Grace, Nikolaos Georgantas, Amel Bennaceur, Gordon S. Blair, Franck Chauvel, Valérie Isarny, Massimo Paolucci, Rachid Saadi, Bertrand Souville, and Daniel Sykes. The CONNECT Architecture. In *Formal Methods for Eternal Networked Software Systems*, volume 6659, pages 27–52. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
- [102] Thomas R. G. Green and Marian Petre. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.
- [103] Object Management Group. CORBA. <http://www.corba.org>.
- [104] Object Management Group. Notification Service Specification Version 1.1. Technical report, Object Management Group, Inc., 2004.
- [105] Tao Gu, Hung Keng Pung, and Da Qing Zhang. A Service-oriented Middleware for Building Context-aware Services. *Journal of Network and computer applications*, 28(1):1–18, 2005.
- [106] Pablo Guerrero, Daniel Jacobi, and Alejandro Buchmann. Workflow Support for Wireless Sensor and Actor Networks. In *4th International Workshop on Data Management for Sensor Networks*, September 2007.
- [107] Jingzhi Guo and Chengzheng Sun. Context Representation, Transformation and Comparison for Ad Hoc Product Data Exchange. In *DocEng'03*, Grenoble, France, 2003.
- [108] David Harel, Assaf Marron, and Gera Weiss. Behavioral Programming. *Communications of the ACM*, 55(7):90, July 2012.

-
- [109] Souleiman Hasan, Edward Curry, and Mauricio Banduk. Toward Situation Awareness for the Semantic Sensor Web: Complex Event Processing with Dynamic Linked Data Enrichment. *Semantic Sensor Networks*, page 60, 2011.
- [110] Annika Hinze, Kai Sachs, and Alejandro Buchmann. Event-based Applications and Enabling Technologies. In *DEBS'09*, 2009.
- [111] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2004.
- [112] Richard Hull, Philip Neaves, and James Bedford-Roberts. Towards Situated Computing. In *Wearable Computers, 1997. Digest of Papers., First International Symposium on*, pages 146–153. IEEE, 1997.
- [113] IBM. WebSphere Business Events.
- [114] Claus Ibsen and Jonathan Anstey. *Camel in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.
- [115] ISO/IEC. EBNF. <http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf>, 1996.
- [116] M Jani, Juha Kela, Esko-Juhani Malm, et al. Managing Context Information in Mobile Devices. *IEEE pervasive computing*, 2(3):42–51, 2003.
- [117] Chamikara Jayalath, Julian James Stephen, and Patrick Eugster. Atmosphere: A Universal Cross-Cloud Communication Infrastructure. In *Middleware 2013*, pages 163–182. Springer, 2013.
- [118] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: A Context-oriented Programming Language with Declarative Event-based Context Transition. In *Proceedings of the tenth international conference on Aspect-oriented software development*, pages 253–264. ACM, 2011.
- [119] Richard M Karp. *Reducibility Among Combinatorial Problems*. Springer, 1972.
- [120] Chris F. Kemerer. Software Complexity and Software Maintenance: A survey of Empirical Research. *Annals of Software Engineering*, 1(1):1–22, December 1995.
- [121] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Longtier, and John Irwin. *Aspect-oriented Programming*. Springer, 1997.
- [122] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A Run-time Assurance Approach for Java Programs. *Formal methods in system design*, 24(2):129–155, 2004.
- [123] Robert Kowalski and Marek Sergot. A Logic-based Calculus of Events. In *Foundations of knowledge base management*, pages 23–55. Springer, 1989.
- [124] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35, April 2010.
- [125] Terry Landers and Ronni L Rosenberg. An Overview of Multibase. In *Distributed systems, Vol. II: distributed data base systems*, pages 391–421. Artech House, Inc., 1986.
- [126] Truong-Giang Le, Olivier Hermant, Matthieu Manceny, Renaud Pawlak, and Renaud Rioboo. Using Event-Based Style for Developing M2M Applications. In *Grid and Pervasive Computing*, pages 348–357. Springer, 2013.

-
- [127] Keunwoo Lee, Anthony LaMarca, and Craig Chambers. HydroJ: Object-oriented Pattern Matching for Evolvable Distributed Systems. In *ACM SIGPLAN Notices*, volume 38, pages 205–223. ACM Press, 2003.
- [128] C. Liebig, M. Cilia, and A. Buchmann. Event Composition in Time-dependent Distributed Systems. In *Cooperative Information Systems, 1999. CoopIS '99. Proceedings. 1999 IFCIS International Conference on*, pages 70–78, 1999.
- [129] David S. Linthicum. *Enterprise Application Integration*. Addison-Wesley Longman Ltd., Essex, UK, UK, 2000.
- [130] David Luckham. *The Power of Events*, volume 204. Addison-Wesley Reading, 2002.
- [131] David C Luckham. *Event processing for business: organizing the real-time enterprise*. John Wiley & Sons, 2011.
- [132] David C Luckham and Brian Frasca. Complex Event Processing in Distributed Systems. Technical report, Stanford University, 1998.
- [133] Kuhanandha Mahalingam and Michael N Huhns. An Ontology Tool For Query Formulation in an Agent-based Context. In *Cooperative Information Systems, 1997. COOPIS'97., Proceedings of the Second IFCIS International Conference on*, pages 170–178. IEEE, 1997.
- [134] Alessandro Margara, Sebastian Frischbier, Tobias Freudenreich, Patrick Eugster, David Eysers, and Peter Pietzuch. ASIA – Application Specific Integrated Aggregation for Publish-Subscribe Systems. Technical report, Technical report, 2011. <http://www.cs.otago.ac.nz/staffpriv/dme/asia/ASIA2011.pdf>.
- [135] Nelson Matthys, Christophe Huygens, Danny Hughes, Sam Michiels, and Wouter Joosen. A Component and Policy-based Approach for Efficient Sensor Network Reconfiguration. In *Policies for Distributed Systems and Networks (POLICY), 2012 IEEE International Symposium on*, pages 53–60. IEEE, 2012.
- [136] Deborah L McGuinness, Frank Van Harmelen, et al. OWL Web Ontology Language Overview. *W3C Recommendation*, 10(10):2004, 2004.
- [137] Nenad Medvidovic, Eric M. Dashofy, and Richard N. Taylor. The Role of Middleware in Architecture-Based Software Development. *International Journal of Software Engineering and Knowledge Engineering*, 13(04):367–393, August 2003.
- [138] Nikunj R Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a Taxonomy of Software Connectors. In *Proceedings of the 22nd international conference on Software engineering*, pages 178–187. ACM, 2000.
- [139] David L Mills. On the Accuracy and Stability of Clocks Synchronized by the Network Time Protocol in the Internet System. *ACM SIGCOMM Computer Communication Review*, 20(1):65–75, 1989.
- [140] Ronald Moen and Clifford Norman. Evolution of the PDCA Cycle, 2006.
- [141] P. A. Muckelbauer and V. F. Russo. Lingua Franca: An IDL for Structural Subtyping Distributed Object Systems. In *COOTS*, 1995.
- [142] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-based Systems*, volume 1. Springer, 2006.

-
- [143] Gero Mühl, Andreas Ulbrich, and K Herrman. Disseminating Information to Mobile Clients Using Publish-Subscribe. *Internet Computing, IEEE*, 8(3):46–53, 2004.
- [144] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A Nominal Theory of Objects with Dependent Types. In *ECOOP 2003–Object-Oriented Programming*, pages 201–224. Springer, 2003.
- [145] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus: An Architecture for Extensible Distributed Systems. In *SOSP’93*, Asheville, North Carolina, United States, 1993.
- [146] Michael Olson. Cloud Computing Services for Seismic Networks. 2014.
- [147] OMG. XML Metadata Interchange (XMI). <http://www.omg.org/spec/XMI/ISO/19509/PDF/>, April 2014.
- [148] Oracle. Event Processing. <http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html>.
- [149] Christine Parent and Stefano Spaccapietra. Issues and Approaches of Database Integration. *Communications of the ACM*, 41:166–178, 1998.
- [150] Adrian Paschke, Paul Vincent, Catherine Moxey, Martin Hirzel, and Alex Alves. Event Processing Reference Architecture – Design Patterns, 2009. Content by: members of the EPTS Reference Architecture Working Group.
- [151] Jason Pascoe. Adding Generic Contextual Capabilities to Wearable Computers. In *Wearable Computers, 1998. Digest of Papers. Second International Symposium on*, pages 92–99. IEEE, 1998.
- [152] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Context Aware Computing for the Internet of Things: A Survey. *Communications Surveys & Tutorials, IEEE*, 16(1):414–454, 2014.
- [153] Milenko Petrovic, Ioana Burcea, and Hans-Arno Jacobsen. S-ToPSS: Semantic Toronto Publish/-Subscribe System. In *Proceedings of the 29th international conference on Very large data bases - Volume 29, VLDB ’03*, pages 1101–1104, Berlin, Germany, 2003. VLDB Endowment.
- [154] Peter R Pietzuch and Jean M Bacon. Hermes: A Distributed Event-based Middleware Architecture. In *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, pages 611–618. IEEE, 2002.
- [155] Daniel Popescu, Joshua Garcia, Kevin Bierhoff, and Nenad Medvidovic. Impact Analysis for Distributed Event-based Systems. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 241–251. ACM Press, 2012.
- [156] Davy Preuveneers, Jan Van den Bergh, Dennis Wagelaar, Andy Georges, Peter Rigole, Tim Clerckx, Yolande Berbers, Karin Coninx, Viviane Jonckers, and Koen De Bosschere. Towards an Extensible Context Ontology for Ambient Intelligence. In *Ambient intelligence*, pages 148–159. Springer, 2004.
- [157] Ragunathan Raj Rajkumar, Insup Lee, Lui Sha, and John Stankovic. Cyber-physical Systems: The Next Computing Revolution. In *Proceedings of the 47th Design Automation Conference*, pages 731–736. ACM, 2010.
- [158] Andry Rakotonirainy, Jaga Indulska, Seng Wai Loke, and Arkady Zaslavsky. Middleware for Reactive Components: An Integrated Use of Context, Roles, and Event-based Coordination. In *Middleware 2001*, pages 77–98. Springer, 2001.

-
- [159] Florian Reimold. Semantic event generation exemplified in a smart home. Bachelor's thesis, Technische Universität Darmstadt, 2013.
- [160] S. P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, 7(4):57–66, 1990.
- [161] Tom Rodden, Keith Cheverst, K Davies, and Alan Dix. Exploiting Context in HCI Design for Mobile Systems. In *Workshop on human computer interaction with mobile devices*, pages 21–22. Citeseer, 1998.
- [162] Henriette Röger and Tobias Freudenreich. Vereinfachung der Ontologierstellung durch Designmuster. Technical report, TU Darmstadt, October 2014.
- [163] Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H Campbell, and Klara Nahrstedt. A Middleware Infrastructure for Active Spaces. *IEEE pervasive computing*, 1(4):74–83, 2002.
- [164] Dan Rosanova. *Microsoft BizTalk Server 2010 Patterns*. Packt Publishing Ltd, 2011.
- [165] Eleanor Rosch. Principles of Categorization. *Concepts: core readings*, pages 189–206, 1999.
- [166] David S. Rosenblum and Alexander L. Wolf. A Design Framework for Internet-scale Event Observation and Notification. *SIGSOFT Softw. Eng. Notes*, 22(6):344–360, November 1997.
- [167] Arnon Rosenthal, Upen S Chakravarthy, Barbara Blaustein, and José Blakely. Situation Monitoring for Active Databases. In *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 455–464. Morgan Kaufmann Publishers Inc., 1989.
- [168] Ioanna Roussaki, Maria Strimpakou, Nikos Kalatzis, Miltos Anagnostou, and Carsten Pils. Hybrid Context Modeling: A Location-based Scheme Using Ontologies. In *Pervasive Computing and Communications Workshops, 2006. PerCom Workshops 2006. Fourth Annual IEEE International Conference on*, pages 6–pp. IEEE, 2006.
- [169] Szabolcs Rozsnyai, Josef Schiefer, and Alexander Schatten. Concepts and Models for Typing Events for Event-based Systems. In *DEBS'07*, Toronto, Ontario, Canada, 2007.
- [170] Tuukka Ruotsalo and Eero Hyvönen. An Event-Based Approach for Semantic Metadata Interoperability. In *The Semantic Web*, volume 4825, pages 409–422. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
- [171] Giovanni Russello, Leonardo Mostarda, and Naranker Dulay. A Policy-based Publish/Subscribe Middleware for Sense-and-React Applications. *Journal of Systems and Software*, 84(4):638–654, 2011.
- [172] Nick S Ryan, Jason Pascoe, and David R Morse. Enhanced Reality Fieldwork: The Context-aware Archaeological Assistant. In *Computer applications in archaeology*. Tempus Reparatum, 1998.
- [173] Kai Sachs, Samuel Kounev, Jean Bacon, and Alejandro Buchmann. Performance Evaluation of Message-oriented Middleware Using the SPECjms2007 Benchmark. *Performance Evaluation*, 66(8):410–434, August 2009.
- [174] Prashanth Sadanand. Distributing components in an event enrichment middleware. Master's thesis, Technische Universität Darmstadt, May 2015.
- [175] Ansgar Scherp, Thomas Franz, Carsten Saathoff, and Steffen Staab. F – A Model of Events Based on the Foundational Ontology DOLCE+DnS Ultralight. In *K-CAP'09*, 2009.

-
- [176] Josef Schiefer, Szabolcs Rozsnyai, Christian Rauscher, and Gerd Saurer. Event-driven Rules for Sensing and Responding to Business Situations. In *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, DEBS '07, pages 198–205, New York, NY, USA, 2007. ACM.
- [177] Bill Schilit, Norman Adams, and Roy Want. Context-aware Computing Applications. In *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on*, pages 85–90. IEEE, 1994.
- [178] B.N. Schilit and M.M. Theimer. Disseminating Active Map Information to Mobile Hosts. *Network, IEEE*, 8(5):22–32, Sept 1994.
- [179] Kay-Uwe Schmidt, Darko Anicic, and Roland Stühmer. Event-driven Reactivity: A Survey and Requirements Analysis. In *3rd International Workshop on Semantic Business Process Management*, pages 72–86, 2008.
- [180] Fred B. Schneider. Synchronization in Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 4(2):125–148, April 1982.
- [181] Stan Schneider. DDS and the Future of Complex, Distributed Data-Centric Embedded Systems. Real Time Innovations, Inc., 2006. <http://www.embedded.com/design/connectivity/4025967/DDS-and-the-future-of-complex-distributed-data-centric-embedded-systems>.
- [182] Len Seligman, Peter Mork, Alon Halevy, Ken Smith, Michael J Carey, Kuang Chen, Chris Wolf, Jayant Madhavan, Akshay Kannan, and Doug Burdick. OpenII: An Open Source Information Integration Toolkit. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1057–1060. ACM, 2010.
- [183] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level Programming Language Design for Distributed Computation. page 15. ACM Press, 2005.
- [184] Lui Sha, Sathish Gopalakrishnan, Xue Liu, and Qixin Wang. Cyber-physical Systems: A New Frontier. In *Machine Learning in Cyber Trust*, pages 3–13. Springer, 2009.
- [185] Chetan Shankar and R. Campbell. A policy-based management framework for pervasive systems using axiomatized rule-actions. In *Network Computing and Applications, Fourth IEEE International Symposium on*, pages 255–258, July 2005.
- [186] J. Singh, L. Vargas, J. Bacon, and K. Moody. Policy-based Information Sharing in Publish/Subscribe Middleware. In *Policies for Distributed Systems and Networks, 2008. POLICY 2008. IEEE Workshop on*, pages 137–144, June 2008.
- [187] Jatinder Singh and Jean Bacon. SBUS: A Generic, Policy-enforcing Middleware for Open Pervasive Systems. *University of Cambridge Computer Laboratory Technical Report TR*, 850, 2014.
- [188] John Skovronski and Kenneth Chiu. An Ontology Based Publish Subscribe Framework. In *ii-WAS'06*, 2006.
- [189] Bruce Snyder, Dejan Bosnanac, and Rob Davies. *ActiveMQ in Action*. Manning, 2011.
- [190] J Soberg, Vera Goebel, and Thomas Plagemann. Commonsens: Personalisation of Complex Event Processing in Automated Homecare. In *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2010 Sixth International Conference on*, pages 275–280. IEEE, 2010.

-
- [191] Standard Performance Evaluation Corporation (SPEC). SPECjms2007 industry-standard benchmark, <http://www.spec.org/jms2007/>, 2007.
- [192] Neville A Stanton, PRG Chambers, and John Piggott. Situational Awareness and Safety. *Safety science*, 39(3):189–204, 2001.
- [193] Mark-Oliver Stehr, Minyoung Kim, and Carolyn Talcott. Toward Distributed Declarative Control of Networked Cyber-physical Systems. In *Ubiquitous Intelligence and Computing*, pages 397–413. Springer, 2010.
- [194] Heiner Stuckenschmidt and Holger Wache. Context Modeling and Transformation for Semantic Interoperability. In *KRDB*, pages 115–126, 2000.
- [195] Kevin J. Sullivan and David Notkin. Reconciling Environment Integration and Software Evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, July 1992.
- [196] Sun Microsystems. Core J2EE Patterns – Data Access Object. <http://www.oracle.com/technetwork/java/dataaccessobject-138824.html>, 2002.
- [197] Carolyn Talcott. Cyber-physical Systems and Events. In Martin Wirsing, Jean-Pierre Banâtre, Matthias Hözl, and Axel Rauschmayer, editors, *Software-Intensive Systems and New Computing Paradigms*, volume 5380 of *Lecture Notes in Computer Science*, pages 101–115. Springer Berlin Heidelberg, 2008.
- [198] Carolyn L Talcott. Policy-based Coordination in PAGODA: A Case Study. *Electronic Notes in Theoretical Computer Science*, 181:97–112, 2007.
- [199] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro P Buchmann. A Peer-to-peer Approach to Content-based Publish/Subscribe. In *DEBS’03*, 2003.
- [200] R. Thurlow. RPC: Remote Procedure Call Protocol Specification Version 2. Technical report, IETF, 2009. <https://tools.ietf.org/html/rfc5531>.
- [201] TIBCO. StreamBase CEP. <http://www.tibco.com/products/event-processing/complex-event-processing>.
- [202] M.M. Trivedi, T.L. Gandhi, and K.S. Huang. Distributed Interactive Video Arrays for Event Capture and Enhanced Situational Awareness. *Intelligent Systems, IEEE*, 20(5):58–66, Sept 2005.
- [203] Sudha Verma, Sarah Vieweg, William J Corvey, Leysia Palen, James H Martin, Martha Palmer, Aaron Schram, and Kenneth Mark Anderson. Natural Language Processing to the Rescue? Extracting "Situational Awareness" Tweets During Mass Emergency. In *ICWSM*, 2011.
- [204] Sarah Vieweg, Amanda L Hughes, Kate Starbird, and Leysia Palen. Microblogging During Two Natural Hazards Events: What Twitter May Contribute to Situational Awareness. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1079–1088. ACM, 2010.
- [205] Steve Vinoski. Advanced Message Queuing Protocol. *IEEE Internet Computing*, 10(6):87–89, November 2006.
- [206] Pieter Vromant, Danny Weyns, Sam Malek, and Jesper Andersson. On Interacting Control Loops in Self-adaptive Systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 202–207. ACM, 2011.

-
- [207] Florian Waas, Robert Wrembel, Tobias Freudenreich, Maik Thiele, Christian Koncilia, and Pedro Furtado. On-demand ELT Architecture for Right-time BI: Extending the Vision. *International Journal of Data Warehousing and Mining (IJDWM)*, 9(2):21–38, 2013.
- [208] Holger Wache and Heiner Stuckenschmidt. Practical Context Transformation for Information System Interoperability. In Varol Akman, Paolo Bouquet, Richmond Thomason, and Roger Young, editors, *Modeling and Using Context*, volume 2116, pages 367–380. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.
- [209] Edward Waltz, James Llinas, et al. *Multisensor Data Fusion*, volume 685. Artech house Boston, 1990.
- [210] Jinling Wang, Beihong Jin, and Jing Li. An Ontology-based Publish/Subscribe System. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, Middleware '04, pages 232–253, New York, NY, USA, 2004. Springer-Verlag New York, Inc.
- [211] Andy Ward, Alan Jones, and Andy Hopper. A New Location Technique for the Active Office. *Personal Communications, IEEE*, 4(5):42–47, 1997.
- [212] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance Complex Event Processing Over Streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 407–418, New York, NY, USA, 2006. ACM.
- [213] Peng Wu, Rajiv Bhatnagar, Lennie Epshtein, Malini Bhandaru, and Zhongwen Shi. Alarm Correlation Engine (ACE). In *Network Operations and Management Symposium, 1998. NOMS 98., IEEE*, volume 3, pages 733–742. IEEE, 1998.
- [214] Alex Wun and Hans-Arno Jacobsen. A Policy Management Framework for Content-based Publish/Subscribe Middleware. In *Middleware 2007*, pages 368–388. Springer, 2007.

Appendix

Study Help Text

DPL

The following code defines concepts, which are used in conditions. If all conditions are met, the defined actions are taken.

Esper/Oracle EPL

The following code is in the event processing language (EPL). It processes events and combines them to new event streams. In this case, the occurrence of an AlarmEvents causes the alarm to be sounded.

Java

The following code is written in Java.

Code

Security Scenario

DPL

```
concepts
  person A (status='guest')
  person B (status='employee')
  room R (security='restricted')
conditions
  A is inside R
  A is not within 5m of B or B is not inside R
actions
  raise alarm
```

Esper EPL

```
insert into UserStream
  select * from pattern[timer:interval(0)],
  sql:db1 ['select * from rfid_tags']

insert into RoomStream
  select * from pattern[timer:interval(0)],
  sql:db1 ['select * from room']
```

```

insert into EmployeeEvent
  select p.id, p.position
  from PersonPositions as p, UserStream as u
  where u.id = p.id
        and u.status = 'employee'

insert into GuestEvent
  select p.id, p.position
  from PersonPositions as p, UserStream as u
  where u.id = p.id
        and u.status = 'guest'

insert into EmployeeRoom
  select e.id, e.position, r.id as room
  from EmployeeEvent as e, RoomStream as r
  where Functions.inside(e.position, r.coordinates)

insert into GuestRoom
  select g.id, g.position, r.id as room
  from GuestEvent as g, RoomStream as r
  where Functions.inside(g.position, r.coordinates)
        and r.security = 'restricted'

insert into AlarmEvent
  select a.room, a.id, a.position from pattern
    [every a=GuestRoom -> (timer:interval(5 sec) and not b=EmployeeEvent
      (Functions.within(position, a.position, 5.0) and room = a.room)
    )]

```

Oracle EPL

```

INSERT INTO UserStream
  SELECT * FROM db1('SELECT * FROM rfid_tags')

INSERT INTO RoomStream
  SELECT * FROM db1('SELECT * FROM room')

INSERT INTO EmployeeEvents(id, position)
  SELECT P.id, P.position
  FROM PersonPositions P, UserStream U
  WHERE U.id = P.id AND
        U.status = 'employee'

INSERT INTO GuestEvents(id, position)
  SELECT P.id, P.position
  FROM PersonPositions P, UserStream U
  WHERE U.id = P.id AND
        U.status = 'guest'

INSERT INTO EmployeeRoom(id, position, room)
  SELECT E.id, E.position, R.id
  FROM EmployeeEvent E, RoomStream R

```

```

WHERE Functions.inside(E.position, R.coordinates)

INSERT INTO GuestRoom(id, position, room)
SELECT G.id, G.position, R.id
FROM GuestEvent G, RoomStream R
WHERE Functions.inside(G.position, R.coordinates) AND
      R.security = 'restricted'

INSERT INTO AlarmEvent(room, person, position)
SELECT a.room, a.id, a.position
MATCHING EVERY (
    a:=GuestRoom FOLLOWED BY
        NOT b:=EmployeeRoom(Functions.within(position, a.position, 5.0) and room = a.room)
    WITHIN 5 SECONDS)

```

Java

```

List<RoomEvent> guests = new ArrayList<RoomEvent>();
List<RoomEvent> employees = new ArrayList<RoomEvent>();

@Override
public void onMessage(Message message) {
    ObjectMessage castedMessage = (ObjectMessage) message;
    Serializable payload = castedMessage.getObject();

    cleanOldEvents();

    if (payload instanceof PositionEvent) {
        handlePositionEvent((PositionEvent) payload);
    }
}

public void handlePositionEvent(PositionEvent position) {
    String status = "";
    int id = position.getId();
    String query = "SELECT status FROM rfid_tags WHERE id = " + id";
    ResultSet rs = connection.createStatement().executeQuery(query);
    if (rs.next()) {
        status = rs.getString("status");
    }
    if ("employee".equals(status) {
        handleEmployee(position);
    } else {
        handleGuest(position)
    }
}

public void handleGuest(PositionEvent position) {
    String room = null;
    Position pos = position.getPosition();
    String query = "SELECT id, status, coordinates FROM rooms";
    ResultSet rs = connection.createStatement().executeQuery(query);

```

```

while (rs.next()) {
    if (!("restricted".equals(rs.getString("status"))))
        continue;
    if (!(Functions.inside(pos, rs.getObject("coordinates"))))
        continue;
    room = rs.getString("id");
}
if (room == null)
    return;
handleGuestInRoom(position, room);
}

public void handleEmployee(PositionEvent position) {
    String room = null;
    Position pos = position.getPosition();
    String query = "SELECT id, coordinates FROM rooms";
    ResultSet rs = connection.createStatement().executeQuery(query);
    while (rs.next()) {
        if (Functions.inside(pos, rs.getObject("coordinates"))) {
            room = rs.getString("id");
            break;
        }
    }
    if (room == null)
        return;
    handleEmployeeInRoom(position, room);
}

public void handleEmployeeInRoom(PositionEvent position, String room) {
    employees.add(new RoomEvent(position, room));
    List<RoomEvent> newList = new ArrayList<RoomEvent>();
    for (RoomEvent re : guests) {
        if (re.getRoom().equals(room) &&
            Functions.within(position.getPosition(), re.getPosition().getPosition(), 5)) {
            continue;
        }
        newList.add(re);
    }
    guests = newList;
}

public void handleGuestInRoom(PositionEvent position, String room) {
    boolean found = false;
    for (RoomEvent re : employees) {
        if (re.getRoom().equals(room) &&
            Functions.within(position.getPosition(), re.getPosition().getPosition(), 5)) {
            found = true;
            break;
        }
    }
    if (!found) {
        guests.add(new RoomEvent(position, room));
    }
}

```

```

public void cleanOldEvents() {
    Date now = new Date();
    List<RoomEvent> newList = new ArrayList<RoomEvent>();
    for (RoomEvent re : employees) {
        if (re.getTimestamp() >= (now.getTime() - 5000)) {
            newList.add(re);
        }
    }
    employees = newList;
    newList = new ArrayList<RoomEvent>();
    for (RoomEvent re : guests) {
        if (re.getTimestamp() < (now.getTime() - 5000)) {
            Functions.raiseAlarm(re);
        } else {
            newList.add(re);
        }
    }
    guests = newList();
}

```

Logistics Scenario

DPL

```

concepts
    truck T
    substance A (dangerous='yes')
    substance B (dangerous='yes')
conditions
    A is inside T
    B is inside T
    distance(A, B) < minimumDistance(A, B)
actions
    notify headquarters

```

Esper EPL

```

insert into TruckStream
    select * from pattern[timer:interval(0)],
    sql:db1 ['select * from trucks']

insert into SubstanceStream
    select * from pattern[timer:interval(0)],
    sql:db1 ['select * from substances']

insert into DangerousSubstanceEvents
    select p.id, p.position
    from SubstancePositions as p, SubstanceStream as s
    where p.id = s.id
    and s.dangerous = true

```

```

insert into DangerousSubstanceInTruckEvents
  select s.id, s.position, t.id as truck
  from DangerousSubstanceEvents as s, TruckStream as t
  where Functions.inside(s.id, t.id)

insert into NotifyHeadquartersEvents
  select a.id, a.position, a.truck
  from pattern [every a=DangerousSubstanceInTruckEvents ->
    b=DangerousSubstanceInTruckEvents(Functions.minimumDistance(id, a.id)
      < Functions.distance(position, a.position) and truck = a.truck)
    where timer:within(1 sec)]

```

Oracle EPL

```

INSERT INTO TruckStream
  SELECT * FROM db1('SELECT * FROM trucks')

INSERT INTO SubstanceStream
  SELECT * FROM db1('SELECT * FROM substances')

INSERT INTO DangerousSubstanceEvents(id, position)
  SELECT P.id, P.position
  FROM SubstancePositions P, SubstanceStream S
  WHERE P.ID = S.ID AND
    S.dangerous = true

INSERT INTO DangerousSubstanceInTruckEvents(id, position, truck)
  SELECT S.id, S.position, T.id
  FROM DangerousSubstanceEvents S, TruckStream T
  WHERE Functions.inside(S.id, T.id)

INSERT INTO NotifyHeadquartersEvents(id, position, truck)
  SELECT a.id, a.position, a.truck
  MATCHING EVERY (
    a:=DangerousSubstanceInTruckEvents FOLLOWED BY
    b:=DangerousSubstanceInTruckEvents(Functions.minimumDistance(id, a.id)
      < Functions.distance(position, a.position) AND truck = a.truck) WITHIN 1 SECONDS

```

Java

```

List<TruckEvent> dangers = new ArrayList<RoomEvent>();

@Override
public void onMessage(Message message) {
  ObjectMessage castedMessage = (ObjectMessage) message;
  Serializable payload = castedMessage.getObject();

  cleanOldEvents();

  if (payload instanceof PositionEvent) {

```

```

        handlePositionEvent((PositionEvent) payload);
    }
}

public void handlePositionEvent(PositionEvent position) {
    boolean dangerous = false;
    int id = position.getId();
    String query = "SELECT dangerous FROM substances WHERE id = " + id";
    ResultSet rs = connection.createStatement().executeQuery(query);
    if (rs.next()) {
        dangerous = rs.getBoolean("dangerous");
    }
    if (dangerous) {
        handleDangerousSubstance(position)
    }
}

public void handleDangerousSubstance(PositionEvent position) {
    String truck = null;
    String query = "SELECT truck FROM substances WHERE id = " + position.getId();
    ResultSet rs = connection.createStatement().executeQuery(query);
    if (rs.next()) {
        truck = rs.getString("truck");
    }
    if (truck == null)
        return;
    handleDangerousInTruck(position, truck);
}

public void handleDangerousInTruck(PositionEvent position, String truck) {
    for (TruckEvent te : dangers) {
        if (te.getTruck.equals(truck) &&
            Functions.distance(te.getPosition().getPosition(), position.getPosition())
                < Functions.minimumDistance(te.getPosition().getId(), position.getId())) {
            notifyHeadquarters(te);
            return;
        }
    }

    dangers.add(new TruckEvent(position, truck));
}

public void cleanOldEvents() {
    Date now = new Date();
    List<TruckEvent> newList = new ArrayList<TruckEvent>();
    for (TruckEvent te : dangers) {
        if (te.getTimestamp() >= (now.getTime() - 1000)) {
            newList.add(re);
        }
    }
    dangers = newList;
}

```