OTTO VON GUERICKE
UNIVERSITÄT
MAGDEBURG

Abstract Sensor Event Processing to Achieve Dynamic Composition of Cyber-Physical Systems

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von Dipl.-Inform. Christoph Steup

geb. am 29.01.1985 in Halle (Saale)

Gutachterinnen/Gutachter

Prof. Dr. rer. nat. Jörg Kaiser
Prof. Dr. s.c. ETH Kay Uwe Römer
Prof. Dr.-Ing. Christian Diedrich

Magdeburg, den 19.02.2018

Zusammenfassung

Diese Thesis widmet sich dem übergeordneten Ziel Cyber-Physische Systeme (CPS) dynamisch aus Komponenten zusammenzusetzen. CPS sind eine Schlüsseltechnologie für die Industrie 4.0, moderne Robotik und autonom fahrende Fahrzeuge. Jedoch werden die realen Systeme immer komplexer und setzen sich aus immer mehr einzelnen Komponenten zusammen, welche über ein Kommunikationsnetz verbunden sind. Die Herausforderung bei der Entwicklung dieser Systeme ist die Spezifikation und der Austausch der im System vorhandenen Informationen um die vorhandenen Komponenten bestmöglich zu nutzen. Die notwendigen Informationen sind dabei sehr Szenario spezifisch. Diese Arbeit versucht die Konstruktion und Komposition dieser Systeme zu vereinfachen indem eine neue Form der Informationsabstraktion für CPS eingeführt wird: Abstract Sensor Events (ASE). Diese stellen die Kapselung von Sensorinformation in maschinenlesbare Form dar. Die Kapselung vereint die inhärenten Sensordaten mit den notwendigen Kontextdaten. Der Kontext der Information besteht hierbei mindestens aus Zeitpunkt, Position, Ersteller und semantischer Beschreibung. Um die unvermeidbare Unschärfe der Sensorinformationen abzubilden, enthalten ASE eine intervallbasierte Unschärfebeschreibung, der Daten und des Kontextes. Die semantische Beschreibung basiert auf einem Wörterbuch von Attributen, die in den Ereignissen enthalten sein können. Dieses Wörterbuch enthält Attribute, die physikalische Phänomene wie Temperatur, Geschwindigkeit etc. abbilden. Szenerio spezifische Erweiterung dieses Wörterbuchs sind möglich, insbesondere mit Attributen, die physikalische Phänomene nur indirekt abbilden. Das Ziel ist es die einzelnen Komponenten des CPS in die Lage zu versetzen ihre Anforderungen bezüglich der eingehenden Informationen mithilfe von ASE zu spezifizieren, so dass ein zugrundeliegendes Kommunikationssystem diese automatisch ausliefern kann.

Hierzu wurde die Abstract Sensor Event Information Architecture (ASEIA) geschaffen, die es Komponenten ermöglicht sich für relevante ASE zu registrieren und mithilfe einer Filtersprache deren Attribute einzuschränken. ASEIAs Aufgabe ist es die benötigten Ereignisse zur Verfügung zu stellen. Jedoch sind die angeforderten Informationen oft nicht direkt im System verfügbar, sondern müssen erst über Verarbeitungsprozesse aus diesen generiert werden. Um diesen Vorgang automatisch ausführen zu können, greift ASEIA auf eine Wissenbasis von Transformationen zurück die es ermöglichen Ereignissen in andere Ereignisse umzuwandeln. Damit stellen die ASE eine Form von aktiven Schnittstellen dar, die zusätzlich zur Abstraktion ihr Verhalten an die gegebenen Bedingungen anpassen. Um die Transformationen auszuführen greift ASEIA auf vorhandene Broker im Netzwerk zurück die die Informationen auf dem Weg von der Quelle zum Ziel anpassen. Die möglichen Transformationen sind in Kategorien unterteilt und verschiedene Beispiele werden in der Arbeit untersucht. Hierbei werden spezielle Transformationen zur Verringerung der Unscharfe der Ereignisse oder zur Zeitsynchronisation darstellt. Klassische Sensorsignalverarbeitungsprozesse werden in Transformationen überführt und theoretisch und experimentell evaluiert. Die Ergebnisse der Evaluation zeigen ASEIAs Nutzen bei der Entkoppelung der Cyber-Physical-System (CPS) Komponenten bei gleichzeitiger Erhöhung der Anpassungsfähigkeit gegenüber Veränderungen in der vorhandenen Informations- und/oder Komponentenmenge. Zusätzlich wurde durch das Design und die Evaluation ASEIAs Eignung auch für eingebettete System bestätigt. Das ermöglicht den Einsatz auf allen typischen CPS Plattformen was ein wichtiger Aspekt für die zukünftige Nutzung in realen Systemen ist.

Danksagung

Ich wäre ohne die Unterstützung von vielen Leuten nicht in der Lage gewesen diese Arbeit fertig zu stellen. Insbesondere möchte ich mich bei Prof. Kaiser bedanken, der mir durch das Karyon Projekt und seine große Erfahrung dabei geholfen hat ein interessantes Thema zu finden und zu bearbeiten und mich sogar nach seiner Pensionierung weiter betreut hat. Ebenso möchte ich Jun.-Prof. Zug danken, der immer ein offenes Ohr für meine Probleme hatte und mit dem ich jederzeit die Inhalte der Arbeit diskutieren konnte. Prof. Mostaghim möchte ich für ich unendliches Verständnis danken, dass die Arbeit doch signifikant länger in ihrer Fertigstellung gedauert hat als geplant. Sie hat immer das richtige Maß an Druck aufgebaut, mich dabei aber nie überfordert. Prof. Römer möchte ich für seine Bereitschaft danken meine Arbeit vorab zu lesen und mir wichtiges Feedback zu geben, das die Arbeit noch mal ein Stück besser gemacht hat. Abschließend möchte ich mich bei meiner Freundin und meiner Familie bedanken, die diesen durchaus langen Weg mit mir zusammen gegangen sind und immer Verständnis dafür hatten, wenn ich mal wieder keine Zeit für sie hatte.

Contents

List of Figures

List of Tables

Table of Listings

Table of Algorithms

Table of Acronyms

ACC Adaptive Cruise Control

AID Attribute Identificator

API Application Progamming Interface

ASEIA Abstract Sensor Event Information Architecture

ASETE Abstract Sensor Event Transformation Engine

ASE Abstract Sensor Event

AS Abstract Sensor

CAN Controller Area Network

CED Complex Event Detection

CEP Complex Event Processing

CE Complex Event

CPS Cyber-Physical-System

CPU Central Processing Unit

CWS Collision Warning System

DAG Directed Acyclic Graph

DBMS DataBase Management System

DHT Distributed Hash Table

DSL Domain Specific Language

DSMS Data Stream Management System

ECA Event-Condition-Action

EBNF Extended Backus-Naur Form

EID	Event Identificator
FIFO	First-In First-Out
GPS	Global Positioning System
HTTP	Hyper-Text-Transfer-Protocol
IP	Internet Protocol
InfAlg	Inference Algorithm
IoT	Internet of Things
KB	Knowledge Base
LLVM	Low-Level Virtual Machine
MAD	Multiple Add
MANET	Mobile Ad-Hoc Network
MCU	Micro Controller
NURB	Non-Uniform Rational B-Spline
OS	Operating System
p2p	peer-to-peer
PC	Personal Computer
PDC	Park Distance Control
PE	Primitive Event
P/S	Publish/Subscribe
QoC	Quality of Context
QoS	Quality of Service
RAM	Random Access Memory
RDF	Ressource Description Framework
RMI	Remote Method Invocation
ROS	Robot Operating System
RelAlg	Relational Algebra

SDF	Sensor Description Framework
SI	Système international d'unités
SN	Sensor Node
SOA	Service Oriented Architecture
SQL	Structured Query Language
STL	Standard Template Library
tc	Traffic Control
TCP	Transmission Control Protocol
TMP	Template Meta-Programming
ToF	Time of Flight
UDP	User Datagram Protocol
UHCS	Uncertainty-aware Hybrid Clock Synchronization
UTM	Universal Transverse Mercator
UI	User Interface
UML	Unified Modelling Language
VANET	Vehicular Area Network
VM	Virtual Machine
V-Rep	Virtual Robotik Experimentation Platform
WRelAlg	Windowed Relational Algebra
WSN	Wireless Sensor Network
XML	eXtensible Markup Language

1. Introduction

1.1. Motivation

Since the beginning of the industrial revolution mankind has strived towards more and more automation to facilitate the daily lives of people. The first major problem that needed to be solved was the continuous supply with energy. One big step towards a solution was James Watt's steam engine, which was patented in 1781. The next major problem was the transportation of goods and people. Carl Benz's 1886 patented car with a combustion engine paved the way for a new individual mobility. Over time cars as well as factories grew more and more complex, which increased the difficulty for people to handle and control them. This third major problem is currently being tackled with the integration of autonomous control system employing sensors and actuators together with digital and analogue control systems to automatically adapt the systems to the environment. So far, all these systems are statically designed and built. Current visions such as Internet of Things (IoT) as described by Mattern and Floerkemeier [109] and Industry 4.0 as described by Brettel et al. [39] promise a dynamic composition and adaptation of entities in a digital world. However, the new possibilities are accompanied by a multitude of new challenges. One of this is the acquisition, management and fusion of sensor data for the various control systems within the autonomous entities. Nowadays, the plethora of available sensors and their individual characteristics are handled by experts designing the systems, but for dynamic systems this approach is not feasible any more. The systems of the future will need to construct themselves out of the available components and data depending on the current state of the environment. This necessitates a different view on development process of the individual components as well as the technology used to implement and connect them. Therefore, this thesis investigates the design, abstraction and combination of independent, maximally decoupled components fetching, managing and processing sensor data within CPS using a special information architecture: Abstract Sensor Events.

To show the challenges this thesis tackles, three example scenarios are sketched in the next Section 1.2. These are used throughout the thesis.

1.2. Example Scenarios

The complexity of CPS is very hard to grasp. Therefore, the following example scenarios provide an introduction to the challenges that may arise in real-word applications. The different scenarios show the different challenges of energy and resources efficiency, dynamic integration of heterogeneous sensor information and tracking of information quality.

1.2.1. Distributed Event Detection: Wireless Sensor Network (WSN) to Detect Forest Fires

WSNs are considered to be the foundation of the IoT. On the one hand, they provide means to acquire, disseminate and partially process sensor data in large areas. On the other hand, Römer and Mattern [134] describe the vast design space of WSN which creates multiple challenges in the development of generic solutions. Therefore, the thesis will use a special applications to discuss the application of the concepts to WSN: the detection of fires within large forests. This application is highly relevant since forest fires cause large monetary and humanitarian damages every year. Consequently, an early detection is a desirable goal. However, there are multiple challenges in building such a system in a reliable and cost efficient way as described by Yu et al.[169].

All envisioned systems consist of Sensor Node (SN)s containing sensors as well as a Micro Controller (MCU) and a communication interface. These are distributed over a large area and build a meshed network acquiring sensor readings and disseminating them towards a sink. The sink processes the data and forwards the results to an user.

A major concern regarding these systems are costs, maintenance overhead and reliability. An expensive system or one needing lots of maintenance or outputs wrong information will be rejected by human operators. This requires the deployed SNs to use low-power embedded hardware to minimize cost and maximize life-time. However, such low-power embedded hardware necessitates a special development process, in which all parts of the system are adapted towards the resource constraints of the hardware.

Typically delay requirements are relatively lax for WSN since the final consumer of the information is generally a human user. This paves the way for the optimization of the system towards efficiency to decrease resource consumption and increase battery life time. Especially communication is extremely expensive compared to computation as described by Sadler and Martonosi [139] and needs to be minimized to extend SN life-time.

Even though it is possible to build static systems implementingthe exactly needed behavior to detect forest fires, it might be beneficial to support multiple sinks subscribing for the fire information or even derived information. Two examples of such additional sinks are fire fighter teams, who want to track down the detected fire. They are interested in two aspects, their distance and direction towards the fire as well as the current speed of the fire. This data can easily be derived from the raw data of the sensor nodes. However, a computation at the global sink and forwarding of this information might be not ideal. Long information paths in sensor networks create large latencies as well as a high unreliability of data delivery. Both are drawbacks for the usage by fire fighters, who need actual and reliable data.

One solution might be an intelligent event based sensor abstraction. This enables processing of information within the network and direct delivery to the mobile equipment of the fire fighters. In consequence more robust and more actual information are delivered to them by reusing information already existing in the network. If the additional data paths are only used when an actual fire is detected the additional energy consumption might be negligible.

1.2.2. Semi-Autonomous Systems: Vehicular Collision Warning System

Since the development of Carl Benz's patented "Motorwagen" a lot has changed. Cars nowadays focus not only on speed and efficiency, but also on safety and comfort. Currently, the car manufacturers are trying to include more and more driving assistance systems to support the driver. Some of them such as the Adaptive Cruise Control (ACC) focuses on the comfort of the driver, while others like side-assist focus on safety. One such system is the Collision Warning System (CWS), which tries to warn the driver of imminent dangers on the road. Nowadays, it is implemented using local sensors equipped in the car perceiving the direct surroundings. Depending on the sensory setup it can detect imminent collisions at the front or back of the car or warn because an action the driver initiates is not considered safe. The amount of situations the assistance system can detect depends on the cars' actual sensor setup. To increase the usability of the system more environmental perception is necessary. One way to acquire additional information is the communication with surrounding cars. Therefore, standard organizations strive towards car-2-car and car-2-roadside communication standards [11],[4]. These standards will provide manufacturers means of communication that are especially tailored towards vehicle needs and allow transmission of information directly between adjacent cars.

However, from an application point of view this is not enough. The standards only define the basic layer of communication and the allowed message types and formats, which include no semantic information. To allow cooperation between multiple cars of different manufacturers a basic semantic abstraction is necessary. An example scenario is the detection of an overtaking car on a highway as described in [157]. The cars in such a scenario might not all be networked and some of them need to be treated as moving obstacles, which need to be detected by surrounding cars. This scenario is depicted in Fig 1.1.

An abstraction of the individual cars' sensor data into an event based sensor abstraction might unify the available sensor data. This enables the use of existing control algorithms as described in [160]. However, the high requirements towards reliability necessary for such safety critical systems hinder the integration of general event systems. Consequently, the communicated data can only be used as an extension of an existing local solution. Each car still needs to have its own CWS. Even though a distributed CWS might be very difficult to create, the individual cars can still exploit the sensors of surrounding cars to extend their local view of the road.

To allow the integration of the data of another car, the local CWS needs to convert the received sensor data into its local representation and assess its trustworthiness as well as its usefulness. Without this step a trustworthy warning can neither be issued nor suppressed by the system.

Finally, the real-time requirements of such a systems are very tight. Classical Internet Protocol (IP)-based approaches might not even fulfil the necessary soft-real time requirements. However, compared to the WSN scenario battery life-time as well as hardware resources and communication bandwidth are not as limited. Additionally, roads are a well structured environments, which eases the development of specially tailored events representing environmental situations.

In summary, an appropriate event system abstracting the individual cars sensor information may support the development an "Internet of Cars", which is able to share the view of surrounding cars to increase the safety and comfort of the passengers.

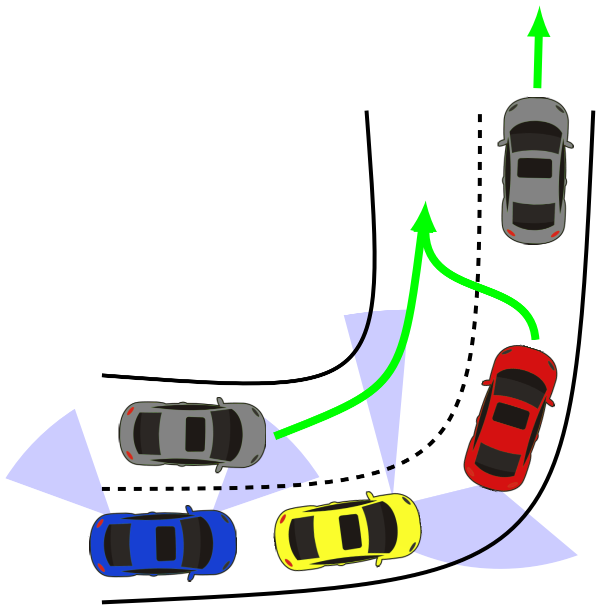


Figure 1.1.: An example scenario of a CWS system detecting overtaking cars on a highway. The overtaking car is not able to communicate and needs to be detected by surrounding cars.

1.2.3. Fully-Autonomous Systems: Dynamic Robot Navigation

Currently, a lot of research is conducted in the area of autonomous robots. Especially, in the context of Industry 4.0 autonomous cooperating robots are envisioned as a key technology for future factories. The logistics companies already review autonomous flying delivery systems to redefine their logistic processes.

In the consumer market the second and third generation of autonomous cleaning robots has reached the markets. These robots are still very simple in their behavior, which is due to their very limited sensory equipment. The Scooba wet floor cleaning robot of the company iRobot has only a front bumper and an odometry sensor for each wheel together with a detector for infra-red beams.

In the talk [152] an example scenario was depicted in which robots need external information to enhance their navigation. In this idea robots should pre-adjust their trajectory to avoid dynamic obstacles. The obstacles could be other robots, objects or persons. Depending on the type of object the acquisition of the necessary information is vastly different. However, all data paths could be grasped with a single event-based sensor abstraction. An illustration of the scenario is visible in Fig 1.2. The image shows two robots R_1 and R_2 in a hallway. The robots are incapable of seeing each other, but there are additional sensors in the environment which extend the perception of them. The goal of this scenario is to enable an awareness of each robot towards the other to enable them to change their navigation plan before they actually see each other. This leads to a better performance than purely reactive navigation strategies. This scenario becomes increasingly complex if the robots are heterogeneous or if one of them is incapable of communication.

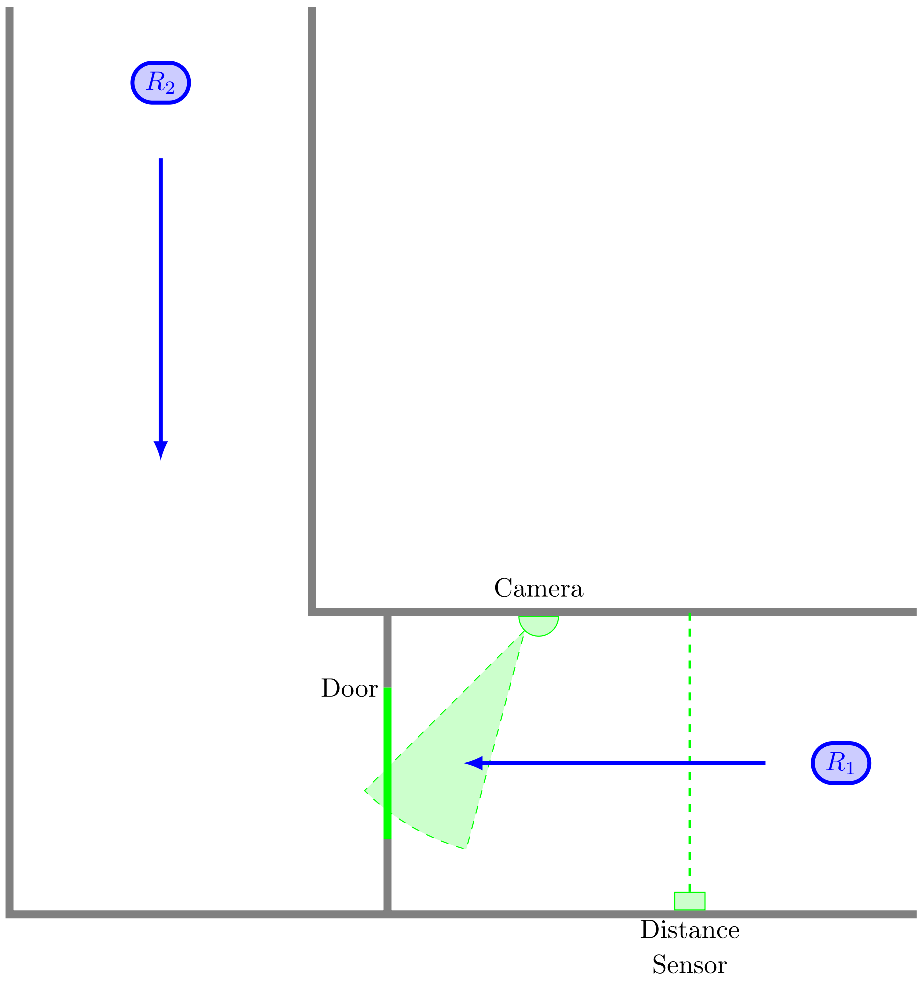


Figure 1.2.: Illustration of a mobile robot avoiding dynamic obstacles based on sensor data from a wireless sensor network.

In future, more autonomous robots will be present in the factories and in the consumers' homes, which will benefit from a mechanism to efficiently share their information. This enables a better perception of the environment for these robots and in consequence enable more complex behavior. Even though current systems are able to handle singular autonomous robots very well, multiple robots acting together is still a big task. Additionally, the robots are similar to the cars, described in Section 1.2.2, very heterogeneous because of different sensory equipment. The necessary parameters of these sensors are relevant for the signal processing and the semantics of the sensor information. An event-based specialized sensor abstraction could be a language enabling the sharing of environment perception between robots without pre-defined messages including additional parameters of the underlying sensors. This might provide a viable solution for multiple robots acting in unknown environments with heterogeneous and possibly unknown sensory equipment.

1.3. Goals

The thesis develops the concept of ASE as an abstraction between CPS components to fullfil the two following main goals, which were motivated in the CPS scenarios described in Sections 1.2.1-1.2.3:

Dynamic Compositon CPS are inherently distributed and components may fail or move out-of-range during run time. This forces the system to adapt by changing the internal structure. The thesis eases this process by dynamically routing connections between CPS components at run-time based on applications needs.

Independent Development The thesis aims to provide ASE as an abstraction to allow an individual development of CPS components. This eases the development process and allows for faster and more reliable production of CPS.

To enable these two main goals, two conceptual sub-goals need to be fulfilled:

Structured Description of Exchanged Information CPS need more explicit data representing the implicit information to handle real-world interaction than classical general purpose software does. Therefore, the thesis investigates the necessary description of this information using machine-readable data structures exchanged between components of CPS. Finally, this information is generalized into an ASE interface.

In-Network Processing of Information Independently developed CPS components that are connected on run-time need interface software mapping their respective possibly incompatible interfaces. The thesis provides these software mappings through the ASE transformations on run-time. The transformations are designed to be distributed in the network to prevent single-points of failure and provide load balancing and scalability.

Additionally, two requirements are necessary to enable the application of the described concept to generic CPS systems:

Flexible Communication Infrastructure CPS need to adapt to the environment, they are used in. To achieve this adaption, a communication is necessary, which establishes communication regarding component requirements. The communication backbone needs to automatically track existing information providers and create links between components depending on the description of the required and provided interfaces of the CPS components.

Compatibility to Embedded Systems CPS are composed of multiple hardware components that range from low-power embedded systems to high-power general purpose processing architectures. The thesis aims to conceptually support a maximum amount of them to maximize the applicability to currently existing CPS. Therefore, resource consumption regarding processing power, memory consumption and network communication bandwidth needs to be considered and evaluated in the design of the system.

2. Challenges of Dynamically Composed Cyber-Physical-System (CPS)

This chapter analyzes, based on the example systems presented in Section 1.2, the challenges that need to be tackled to successfully decouple individual components of a CPS. As a starting point CPS are defined and the general structure of a CPS are described based on Figure 2.3. Afterwards, the individual challenges are described based on the functionality and the components of the CPS. Finally, an overview of additional design challenges is provided, assuming that the components and their functionality are developed in a distributed way.

2.1. Definition: Cyber-Physical Systems

A CPS is defined by Baheti and Gill as:

“The term cyber-physical systems (CPS) refers to a new generation of systems with integrated computational and physical capabilities that can interact with humans through many new modalities. The ability to interact with, and expand the capabilities of the physical world through computation, communication, and control is a key enabler for future technology developments.” [33]

Edwar A. Lee describes CPS as:

“Cyber-Physical Systems (CPS) are integrations of computation with physical processes. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa. ” [98]

A disambiguation between a classical embedded system and a CPS is not easy. As described by Arlat et al. [28] a CPS focus on the connection of individual autonomous entities, which together perform four major activities: "data collection (sensors), communication (openess and interaction), information processing (analysis, optimization and decision) and action on the environment". As most of these activities are also present in embedded systems CPS might be viewed as an evolved form of embedded systems. In general, CPS may consist of multiple autonomous components interacting with the real world in a way that is invisible to the user. Additionally, these components are connected using digital communication interfaces.

This definition of CPS is very generic and needs further refinement to be able to deduce generic challenges in the design of these systems. The following section provides some properties that are typical for CPS to enhance the understanding of the challenges.

2.1.1. Properties

Typical CPS need to fullfil functionalities while providing specific properties. Wan et al. [146] defined the following properties:

Integration of software and physical properties in a complex system.

Limited Ressources : The CPS functionality is embedded in every component, especially in the resource constraints ones such as MCUs.

Networking and Scalability : The CPS consists of many components connected with various communication technologies.

Dynamic Reconfiguration : CPS need to adapt themselves to changes in topology or environment.

Autonomy : CPS favor autonomous behavior and closed control loops over human-interaction.

Dependability : Users depend on the delivered functionality, which requires the system to provide guarantees on the quality of its service.

Not all of these properties need to be fullfilled by all components of the CPS. Depending on the functionality of the component different properties are relevant. The individual components need to communicate to work together and form the CPS. The next section discusses the difference between data and information from the perspective of these components and how this difference influences the interface between the components.

2.1.2. Data and Information

CPS are inherently distributed, where each components is focused on processing, interpretation and communicating data. The whole design of CPS is data-driven as no application can exists without at least some sensory input from the real world. Two typical application classes of CPS are control and monitoring. A control application uses incoming sensor data to modify the real world using actuators to fullfil its goal. A monitoring application uses incoming sensor data to output condensed information to an observing user in a periodic or event-driven way. Both system types need to acquire information on the environment through existing sensors, process it and react to it. If the individual components of the CPS are developed by different parties, the used abstractions are different. As a consequence, the data inside one component may be incompatible with the data of another component. To be able to use these components together in a single CPS, the internally used data needs to be enhanced to represent the contained information in a machine-readable way to enable communication between the components without human intervention.

In this thesis the terms data and information are used based on the following definitions:

Data denotes raw digital values abstracting specific parameters of the environment, components or applications.

Information denotes enhanced data that is prepared to be subject to automated processing, distribution and reasoning without additional external knowledge.

Consequently, the generic problem of large scale CPS is the specification of the information as data in a machine-readable way. Currently, this meta-information translating data to information is called *context*. The next section will discuss the structure and possible and mandatory attributes of a generic context.

2.1.3. Context

CPS are deeply integrated in real world processes. Typically, the CPS logic contains a model of the process it is part of, to enable estimation, prediction and detection of abnormal states. This is important because CPS directly influence the real world and the persons living in it. Reliability and trustworthiness are therefore important aspects. To be able to assess this real world state, contextual information in addition to the pure sensory input is necessary. *Context* provides the necessary link between the different low level sensor data, static knowledge and model information to enable the CPS application to reason on its current functional state and maybe reconfigure itself to adapt to changes in the environment, as described by Zug et al. [170].

The context information necessary for a CPS application is very much dependent on the functionality. As a consequence, it may consist of numerous parameters, which describe aspects of acquisitions, processing and delivery of data. Additionally, the context may contain information on temporal and spatial relations between individual observations of sensors and components in general. To describe generic sensor information, at least the following context attributes are necessary:

Time is the most basic parameter because of its relevance for most physical processes. The benefits of having a trustworthy time information reaches from the ability to order the events to the computation of time derivatives. Additionally, it may be used to observe the latency of a system and react to it by e.g. dropping very old data or prioritize recent data. However, the provision of a stable, reliable time base for a distributed system is a challenge itself. There are no perfect solution and the system designer always needs to choose an appropriate mechanism.

Position is almost as relevant as time. It enables filtering of data based on physical locations. Additionally, it enables the deduction of neighborhoods and groups of nodes possibly observing the same phenomenon, which is important for fusion algorithms enhancing data quality and aggregation.

Uncertainty measures the quality of non-faulty data. This enables applications using this data to decide on the usefulness of the individual event towards their goals. It also enables applications to adapt its decisions based on the quality of the incoming data, which is relevant for robust systems. However, providing a trustworthy uncertainty measure is a challenge on its own. Uncertainty exist in two flavors. On the one hand, the existence of the event itself may be uncertain, which is especially relevant for events directly relating to changes in the environment. On the other hand, the attributes of the event limiting the information contained in the event may be uncertain.

Semantic defines the meaning of the sensors data regarding the observed physical phenomena. The same sensor may provide different information depending on the physical process it is used to observe. An example is a distance sensor attached to a car as part of a Park Distance Control (PDC) and the same distance sensor used to detect the current fill level of a tank of fluids. The sensor data is similar, but the information differs strongly. One typical approach handle the information as object observation complexes, which allows disambiguation based on the object. Another approach is the definition of virtual sensors that enhances the raw sensor data with data representing the semantic information and provide virtual semantic sensors.

Quality is typically considered on two levels. The first is Quality of Service (QoS), which describes the current state of a component to fullfil is specification. QoS parameters allow the specification of different metrics, to which a component should optimize its delivered function. An example is the QoS parameter latency, which describes the age of the information provided by a component. Another often used QoS parameter is package loss, which describes the probability of the information to be completely lost. Through the QoS parameters an application is able to specify its needs regarding these parameters to the component and the component shall aim to provide its service with the specified parameters. Another level of quality is Quality of Context (QoC), which describes the quality of the information produced by the component. As information is an interpretable self-contained enhancement of pure data, QoC parameters describe the context in terms of completeness and accuracy. An example is the context of a distance sensor, which typically consists of the position, time and angle of the measurement, together with the uncertainty of the actual distance. A maximum QoC is achieved if all information is provided with maximum accuracy. In general, arbitrary high accuracy is infeasible, an example is a distance sensor which cannot know where in its opening angle the object was detected. This creates inaccuracies in the provided information decreasing the achievable QoC.

CPS can be viewed from two perspectives. The first perspective looks at the functionality of CPS on different levels of abstraction and how the general requirements provide challenges for each layer. The second perspective focuses on the physical topology of the components. This allows a classification of functionality typically present in each component and the necessary interfaces. Both of these views are discusses in the following to extract the challenges of dynamically composed CPS.

2.2. Functional Decomposition of CPS

A CPS combines multiple Layers of functionality reaching from low-level acquisition of data to high-level decision and adaptation processes. A possible 5-Layer architecture is shown in Figure 2.1 proposed by Lee, Bagheri and Kao [100]. This architecture is tailored towards Industry 4.0 [99]-based manufacturing processes. It consists of the following functionalities:

Smart Connection Layer enables Plug&Play communication between the components.

Data-to-Information Conversion Layer analyzes and enhances the data of the low-level sensor to incorporate contextual information.

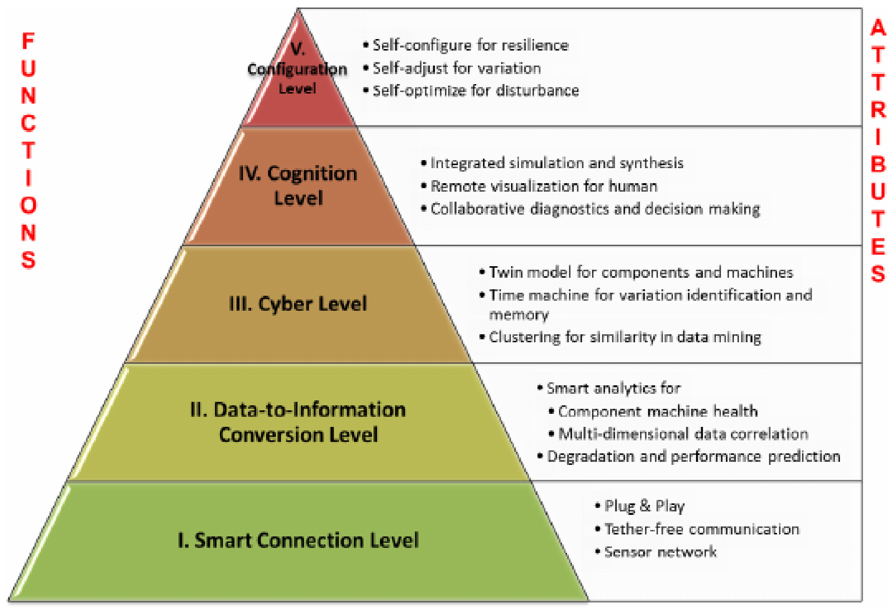

FUNCTIONS
V. Configuration Level
• Self-configure for resilience
• Self-adjust for variation
• Self-optimize for disturbance
IV. Cognition Level
• Integrated simulation and synthesis
• Remote visualization for human
• Collaborative diagnostics and decision making
III. Cyber Level
• Twin model for components and machines
• Time machine for variation identification and memory
• Clustering for similarity in data mining
II. Data-to-Information Conversion Level
• Smart analytics for
• Component machine health
• Multi-dimensional data correlation
• Degradation and performance prediction
I. Smart Connection Level
• Plug & Play
• Tether-free communication
• Sensor network
ATTRIBUTES

Figure 2.1.: Visualization of the 5-Layer CPS functionality architecture by Lee, Bagheri and Kao [100].

Cyber Layer fuses the data from the lower layer to enhance the quality of data and QoC. Additionally, it may inter- and extrapolate data as necessary.

Cognition Layer simulates possible results and enables collaborative diagnostics and decision making.

Configuration Layer adapts the system to environmental changes and allows user interaction and configuration.

2.2.1. Smart Connection Layer

The *Smart Connection Layer* is responsible for coupling the different components forming the CPS dynamically on run-time. To this end, it needs to provide communication abstractions that may be used by higher level functionalities, as well as low-level functionalities to interact with hardware. A sensor component needs to be able to acquire sensor data and distribute them to other components using communication facilities. On the hand , an actuator requires actuation data which is provided by other components through communications facilities to modify the environment accordingly. Some components may not contain any sensing or actuation facilities and are solely responsible for computation and forwarding of communicated messages. Depending on the current task of the system different information flows need to be handled by the *Smart Connection Layer*. During normal operation information needs to be routed from the sensors to the processing and storage components and processed information needs to be distributed from the control applications to the actuators. Additionally, monitoring information is distributed from all components to the user(s). During configuration of the system the information flow is typically backwards, configuration information needs to be distributed back from the actuators to the processing components and from the processing components to the sensor components. Additionally, configuration information may be distributed by the user to the different components of the system. The generic challenges observed in this layer are the following:

Timeliness : Information in a CPS always consists of time-value entities and therefore ages. The timely delivery of information is often crucial for the functionality of the system, but even systems not depending on timeliness delivery at least need information on the age of the information.

Reliability : A reliable communication is generally the baseline of any CPS. Applications for systems, in which information delivery can be guaranteed, are much easier to develop. However, reliability cannot be guaranteed in all cases. Therefore, additional mechanisms need to handle the loss and possible retransmission of the relevant information. In case automatic recovery is not possible, higher layers need to be informed to enable adaptation of the system.

Hardware-Constraints : Depending on the system different amounts of bandwidth and latencies can be achieved. These parameters need to be considered in the design phase of the application. This hinders the development of generic applications unless the applications adapt to varying network context parameters.

As described in Section 2.1.3, pure data is not enough to link components together. The dynamic link needs information to enable automatically composition of the system from existing components. The next section describes the layer that enhances the data with context to information.

2.2.2. Data-to-Information Conversion Level

This layer is responsible to enhance the raw data of the sensors with context to represent the whole information delivered by the sensor. This provides a local view of the environment at a specific location at a specific time for a specific phenomenon. The context attributes are discussed in Section 2.1.3. This functionality is similar to the functional difference between a classic sensor and a *Smart Sensor*. However, classically *Smart Sensor* focus on self-description and self-assessment, while the enhanced CPS sensor components additionally need structured machine-readable context descriptions. The source of the context information is heavily application dependent. In special cases, the necessary context data can be statically compiled in the component. An example is a fixed distance sensor as part of an instrumented environment. However, in most cases the context data need to be acquired through other low-level sensors. Unfortunately, the context of such low-level data can by definition not be complete. Therefore, a CPS need to be able to also express the QoC to disambiguate between fully contextually enhanced sensor data and raw-data, whose context is yet unknown.

This layer may also contain self-testing functionality to add contextual information on reliability of the data. This quality estimation relies on static analyses of the low-level sensors, the results of the self-assessment and possible cross-validation of redundant sensors.

As a consequence, the layer needs to tackle multiple challenges:

Context Information Aquisition : The layer needs to use the *Smart Connection Layer* to acquire the necessary contextual information to provide the maximum amount of context specification to enhance the usability of the data for other components. The necessary contextual information is dependent on the system but at least contains data on time, position and semantics.

Quality Estimation : To disambiguate between the different levels of contextual specification and reliability of context information, the data and context quality need to be estimated to enable other components to handle incomplete or unreliable information.

Semantic Enhancement : To be able to automatically use the acquired data, a machine readable description of the semantics of the information is necessary. This enables higher layer functionality to assess the usability and the compatibility of the information towards their own goals.

As described, the *Data to Information Conversion Layer* cannot always fully define the context. Additionally, it can only estimate the quality of the information, but it does not modify the quality. These two functionalities are typical for the next layer: the *Cyber Layer*.

2.2.3. Cyber Layer

The **Cyber Layer** uses the information created in *Data to Information Conversion Layer* to produce an adequate virtual representation of the real world in cyberspace. To this end, it needs to be able to combine multiple possibly heterogeneous independent information to fullfil the application's requirements on semantic, QoC and QoS. Afterwards, The resulting information can be used again as input for another instance of the *Cyber Layer* or be directly transmitted to the *Cognition Layer* to be used in decision making and control. It may also be forwarded to the *Configuration Layer* for monitoring purposes.

The combination of multiple heterogeneous information is a very broad field of processing with lots of possibilities researched over the years. The field can be separated into three sub-categories.

Selection describes the process of assessing the usefulness of contained information towards the goal of the component and classifying it in two groups of *use* and *don't use*.

Fusion describes the combination of multiple sensor information to a single new sensory output information with higher *quality*.

Reasoning describes the combination of multiple discrete information on the sub-state of the system to new information on the (sub-)state of the system.

Selection

Selection describes the process of estimating the compatibility of available information against the application's goals. This process is very important in CPS, because much information may be generated, but only a small fraction of it may be relevant to achieve the desired functionality. The selection process allows the system to route the relevant information to the individual components, without overloading processing and network links. The selection can work on multiple levels, but generally it can only reference existing context and sensory data. In the most trivial case, hard coded paths may be used based on the actual publishers and subscribers of the information. These hard coded path are for example IP-Address port pairs or Controller Area Network (CAN) identifiers. However, this approach is unsuited for dynamic systems as these IDs are not known on design time. In this case more content-based selection criteria may be used. An example are time- and space-based description routing as done by geo-casting systems, as described by Recksiedler [131]. A third approach is the actual statement of parts of the content of the information as a template to match existing information as done by content-based P/S and most database systems.

Additional, to the type of comparison used in the selection process the type of reference is also important. An example is the comparison of the content-based P/S selection against general database queries. The content-based subscription only allows the comparison against static values, because it typically has no knowledge of other *Events* existing at the same time in the system. On the other hand the database already contains all existing information and can also compare different information.

Therefore, the *Selection* processes can be classified based on the flexibility of comparison operations and regarding the available type of references for these comparisons.

Fusion

Fusion is a general term that describes any combination of incoming sensor information to output sensor information of higher quality. The input sensor information is classified into homogeneous and heterogeneous modality and into overlapping and non-overlapping. The modality in this notion relates to the observed phenomenon type. Sensory information of homogeneous modality observes the same type of phenomenon, whereas heterogeneous modality sensor information observes different types. Overlapping relates to the time space relation of different sensor information. The information overlaps when the time and position of the contained phenomena overlaps. Based on these classification of sensor information, Elmenreich [61] categorized three general types of *Fusion* mechanisms:

Cooperating Fusion combines multiple non-overlapping information of homogeneous modality to a single information with an extended range. A typical example is the combination of multiple cameras with a limited viewing angle to a single camera with an extended viewing angle. The resulting sensor can never be more reliable or more certain than the worst originating one.

Concurrent Fusion combines multiple overlapping information of homogeneous modality to a single information with a higher reliability and/or a smaller uncertainty. A typical example is the combination of three distance sensors facing exactly in the same direction at the same time by majority vote.

Complementary Fusion combines multiple information of heterogeneous modality to a single information of heterogeneous or homogeneous modality. An example is the combination of two distance sensors being orthogonal to each other, that detected the same object at the same time. The fusion can now produce a 2D-position out of these

1D-distances. The combination of the individual Time of Flight (ToF) distances of Global Positioning System (GPS) to a position is also an example.

Actual existing *Fusion* mechanisms are often a combination of these theoretical concepts, especially if the fusion mechanisms was specially designed for an application. In the following, some general examples of fusion operations are described.

Digital Filter uses homogeneous input data and combines them using Multiple Add (MAD)-operations over time. The incoming data needs to originate from different sensors observing the same physical phenomenon or different times of observation, but not both. Since physical phenomena are typically only observable in a limited space, the sensory data needs to stem from the same position. The result is a pure **Concurrent Fusion**. Typical examples are *Digital Low-Pass- and* Digital High-Pass-Filters, which separate the information into a fast changing and a slow changing part and attenuate one of these parts. These general filters can also be used to implement a **Cooperating Fusion** if the input sensory data is not from the same time or from the same position. In this case the quality of the data may be enhanced, but the QoC is decreased as the actual position or time of the observation is also combined.

Kalman-Filters are specialized filters, that combine multiple heterogeneous information sources over time using an estimation of each source's uncertainty. Additionally, the **Kalman-Filter** uses a model of the environmental physical processes to predict the future state of the system. The prediction as well as the different input information are evaluated regarding their uncertainty and combined accordingly. This allows the filter to adapt itself to quality changes in the input information and even provide output in the case of complete omission of input. The **Kalman-Filter** is very flexible and can create any kind of fusion, but is typically used for **Concurrent** and **Complementary** Fusion.

Concatenation stitches different homogeneous data together to form a larger data set. This operation is typically used with multi-dimensional information such as images, maps and point lists. The input information needs to be from the same time or position. An example would be a virtual 360° camera detecting objects in the surroundings. The individual images of the cameras need to be stitched together based on the time of acquisition and their position and orientation. The resulting image can then be evaluated regarding contained objects. This even allows detection of objects, which are partially visible in two adjacent images. The resulting fusion is a typical **Cooperating Fusion**.

Reasoning

Reasoning algorithms provide the CPS with the ability to deduce its current state based on observed (sub-)state changes. The difference to *Fusion* is the focus of these reasoning algorithms on discrete states. In general, reasoning and fusion tackle the same generic problem of either generating new information from existing one or to validate existing information based on other existing information. *Reasoning Algorithms* can be categorized in two major classes: probabilistic reasoning and rule-based reasoning.

Probabilistic Reasoning uses random variables to represent sub-states of components and uses for example **Bayes Theorem** to reason about more general states based on observed information. In general, this type of reasoning considers the state to be discrete or continuous. The reasoning is done in a form of an acyclic graph of inference relations combining the different sub-states in the form of random variables. The graph can grow arbitrarily complex, but the computational effort is exponential limiting realistically usable sizes. The complexity of continuous *Bayesian Networks*, see [93], leads to the development of specialized algorithms to handle typical types of these networks e.g: *Markov Networks* [93] or *Kalman Filter*, see Section 2.2.3. Discrete Bayesian networks can be used to reason on state information in a quality aware way.

Rule-based Reasoning uses exact state information in the form of facts as input and logical deduction rules to infer new state information from existing one. This approach is heavily used in logic programming. Existing logic programming language interpreters use algorithms such as *RETE* [66] to efficiently compute new facts by matching patterns to existing facts and generate new facts as result. The algorithm is typically executed recursively to enable the generation of all possible facts from the existing information.

Challenges

The previous sections described existing mechanisms to process incoming sensory information based on user specified operations. General CPS typically contain multiple types of these operations distributed over multiple components. Fusion operations may be executed directly in the *Smart Sensor*, whereas, *Reasoning* is executed in the application on the mobile device. However, for a dynamically composable CPS the functionality of executing these operations need to be flexible enough to be executed at multiple points in the CPS. Therefore, multiple challenges that need to enable dynamic composition of this layer arise.

Generality : The CPS needs to provide a general approach towards the specification of the necessary operations. The system designer is then able to express his or her needs on the systems functionality.

Dynamic Configuration : The CPS should reconfigure the connections between the individual processing functionalities depending on the current state of the system. Certain fusion types enhancing the quality might not be needed if the quality of the input information is already high.

Quality Estimation : The processing needs to be quality-aware. The quality of the information is relevant for the design of system, but the dynamic composability makes off-line quality computations impossible. In consequence, the quality information needs to be processed and output as well.

Quality Guarantees : Quality requirements stated by the application designer need to be used to configure the necessary processing. This allows the system to adapt itself based on the required and provided quality of the different components.

Context Awareness : The contextual information is very important for the CPS. Therefore the processing cannot limit itself to use the existing contextual information, but also needs to modify it according to the executed operations. An example is the aggregation operation, which typically modifies the time or space context of the information.

After the sensor information was processed to deduce the necessary information on the current state of the system in all continuous and discrete variables, the application may decide on the actuation. This is done in the next layer: the *Cognition Layer*.

2.2.4. Cognition Layer

The *Cognition Layer* contains the functionality, which defines the actual behavior of the CPS. In general, it defines which situations demand which actions by the system. For the robot navigation scenario this represent the actual choice of direction of the robot and for the vehicle warning scenario alarming the user.

The baseline of all these behaviors are applications representing the core of the CPS. Most CPS try to provide the developer of the behavior with abstractions that ease the task of implementing the necessary functionality. These abstractions typically consist of interfaces abstracting away the details of the lower layer, especially of the *Smart Connection* and *Cyber Layer*. The developer of the behavior should only state his or her requirements on the data and implement the necessary applications based on these data. The lower layers shall take care of the fullfilment of these requirements. The requirements differ heavily depending on the application implemented. For the forest fire scenario latency and timeliness are not as important as consistency and accuracy of data. If the fire detection system issues false positives all the time, no one will react to the alarm. In the robotic navigation scenario latency and uncertainty estimation are important, as the robots need to plan ahead depending on the amount and quality of information they receive from the lower layers of the system. In this case the environment is too dynamic to guarantee the fullfilment of requirements. Therefore, the application designer needs to cope with the existing uncertainties. In the vehicle scenario some requirements such as age of information needs to be guaranteed, whereas, others such as position uncertainty may be handled by the application. In consequence, the application needs to be provided with the ability to express requirements to the lower layers and also be informed of the quality of the information it gets from the lower layers. In the following, some example applications are sketched, which lead to the definition of the challenges these present to a dynamically composed CPS.

Closed Loop Control enables the control of physical phenomena relevant for the system. This control needs up-to-date information on the current state of the phenomenon. Also, uncertainties of the observation or errors in the acquisition lead to wrong information fed to the control algorithm and consequently to wrong actuation. This type of application is generally very sensitive to age and quality of information. An example is a Distributed ACC system envisioned for autonomous cars.

Collaborative Decision Making enables CPS consisting of multiple active entities to coordinate their behavior and act as a single entity. An example is the robotic navigation scenario, in which multiple robots coordinate their movement based on the actual position information of each other and the obstacles. The decision making is typically done using communication facilities of the network and therefore needs time to be executed. The timeliness of sensor information is not as important in this case. However, the quality of the sensor information is very important as byzantine errors increase the necessary time for consensus. As a result, an awareness of quality may decrease the time to consensus, even though this speed-up cannot be guaranteed.

Navigation is a generic application in any CPS containing autonomous mobile entities. Depending on the observation capabilities of the system the navigation task has to face different challenges. In general, the aim is to plan a movement for an entity from position A to position B. If the environment is obstacle free, the execution is a simple *Closed Loop Control*. However, if unknown unmoving obstacles may be present, additional sensors are necessary to detect these. Therefore, the navigation needs access to these sensors and it needs to evaluate the quality of the sensors to evaluate the risk of collision based on the requirements. If the obstacle itself is able to move, the challenge can extend to *Collaborative Decision Making* using the current movements as input for both entities trying to evade each other. In this case either timing is very important or the entities adapt their speed.

These generic examples of applications present in CPS lead to the following challenges:

Flexible Control is necessary as the environment of CPS may be dynamic and changes in the environment may not be compensated by lower level components. This requires changes in the behavior of the system as a whole and especially in the functionality of this layer.

Unknown Environments : As the environment is typically not fully known, the layer needs to contain mechanism to evaluate how well the system is suited to the current situation. In case of environment situations in which the quality of the sensor information is to small the system may change its behavior to a fail-safe to prevent harm.

Incomplete Knowledge : Typically, either the context or the information is partially incomplete. The behavioral functionalities present in this layer need to be inherently aware of this lack of information. Additionally, they need to be able to take this lack of information into account, whenever they decide on appropriate actions.

Lack of Guarantees : In dynamic CPS not all requirements can always be guaranteed to be fullfilled. Therefore, the application needs to be able to handle these situations, by requiring lower quality input or degrade its output quality if necessary, as described by the Karyon Architecture [52].

Heterogeneity of Interfaces : The incoming information is delivered by numerous heterogeneous components, this layer needs to hide the heterogeneity to present the application with a uniform interface independent of the informations source.

2.2.5. Configuration Level

The *Configuration Layer* is used to enable the user of the system to monitor and reconfigure the system on runtime. It may also contain automatic reconfiguration components watching over the system and rewiring components if the environment changes. There are two general Use-Cases in this layer. The first consists of automatic information collection on arbitrary states of the system. This is typically configured by a human user if the system shall be evaluated or if some errors where observed and additional information for error mitigation is necessary. To this end, the system needs to provide the user with information on its internal state on request of the user. Depending on the interfaces present in the system this can be achieved either through displays visualizing the information to the user or through the communication interface of the system. The other Use-Case is the reconfiguration of the system. This typically means that the user changes the parameters of the application or the environment changes and a monitoring component reconfigures parameters of the system. An example of such a reconfiguration is the change of the warning distance of the front crash warning system in a car. The user may change this distance because of his or her driving behavior to prevent a high amount of false positives. A sufficiently intelligent system may also be able to learn that its warning distance is to small because of a high amount of false positives and automatically extends the distance to minimize the amount of false positives.

The challenges of this layer are mainly in the changes the layer can induce in other layers. Therefore, the other layers need to be designed to be flexible. The reconfiguration process may change requirements of the system in a way that can only be achieved by changing the underlying communication paths and the executed functionality, especially in the *Cyber Layer*. Additionally, the presentation of the data to the user is difficult, as a high amount of information is present at each time instance in a CPS. Even by using a user supplied filter for the data, the update of information might still be too fast for humans to recognize. Therefore, the layer needs to provide some sort of information fusion mechanism to distill the relevant information for presentation to the user. This may be enabled by correctly configuring the *Cyber Layer* to handle the fusion of the information to be presented.

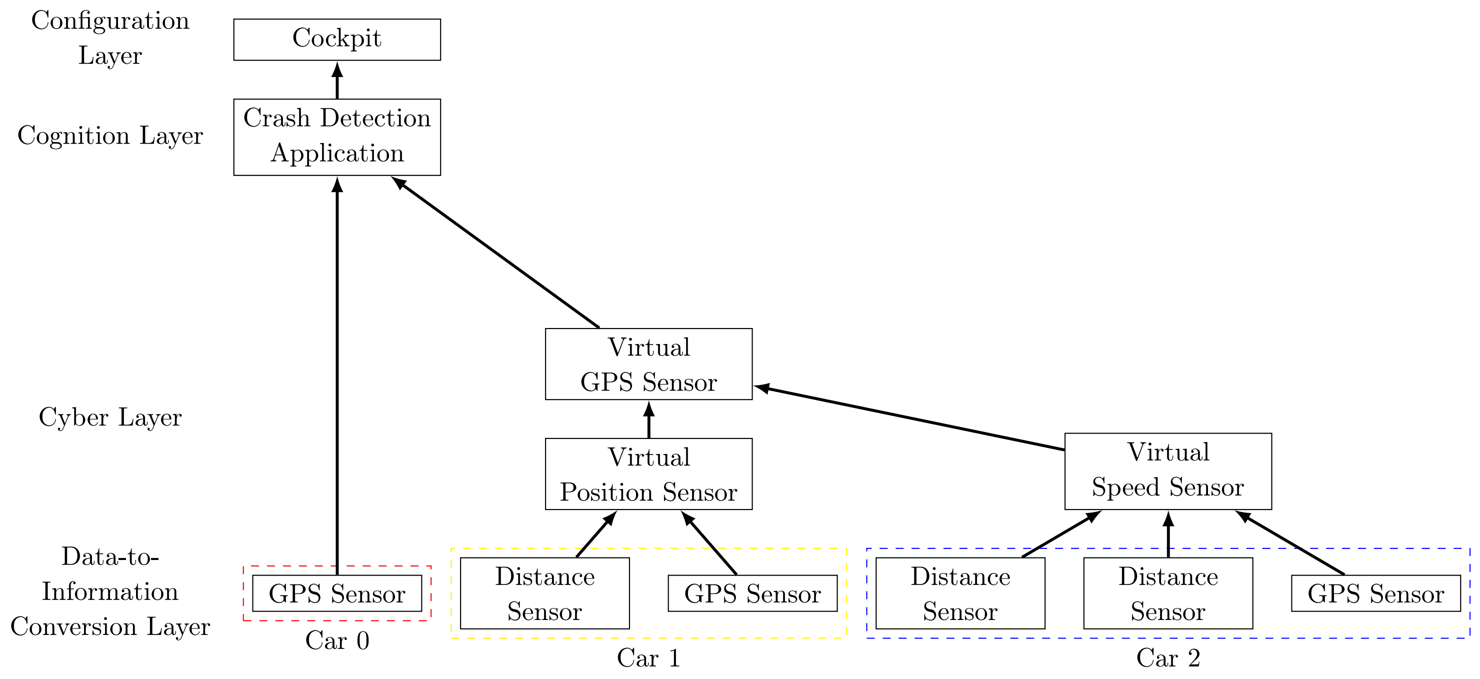


Figure 2.2.: Schematic illustration of the layers used in the CWS example system.

2.2.6. Summary

This layered architecture shows the idea of an increasing level of abstraction and a transition from sensor processing to high-level decisions. On the lowest level of abstraction, raw sensor data and direct actuation commands are handled. On higher layers of abstraction, the sensor data become more an more abstract as do the actuation commands. The processing changes from signal processing to data fusion configured by context to information fusion and inference. On the highest level of abstraction reconfiguration and user control is provided. To present an example of an application split into these layers, the vehicular CWS is depicted in Figure 2.2 showing the functionalities assigned to the different layers.

2.3. Component View of CPS

Generally, CPS consist of multiple nodes connected by a communication network. Each individual node is autonomous and itself composed of at least two components: Processing and Communication. The actuation, sensing, storage and user interface components are optional, but often present. This basic structure is illustrated in Figure 2.3. The tight interaction between these components on different levels of abstraction, see Section 2.2 on different nodes creates the behavior of the CPS .

In the following, the individual components are described in more detail. The characteristic properties are outlined together with some examples for components used in exemplary systems.

Components can be categorized into the following groups:

Sensing components acquire observations of real world phenomena and provide them as digital information to the CPS.

Actuation components use digital information on real world phenomena to change the current state of the real world.

Communication components route and deliver information from one component to another in the CPS.

Processing components modify incoming information into outgoing information.

Storage components allow the permanent or semi-permanent storage of information for later retrieval.

Montoring and Configuration components allow users to interact with the system.

2.3.1. Sensing

CPS may only perceive their environment through sensors. The acquired *Sensor Information* is highly relevant to assess the environment and react to it. The process of acquiring the sensor information is performed by specialized components that together form a *Sensor*. The acquisition starts with the transduction of any kind of physical phenomenon to a digital signal. Afterwards, the digital signal is processed and conditioned. As a result, *Sensor Data* are acquired. To be able to use this data, it needs to be extended with additional

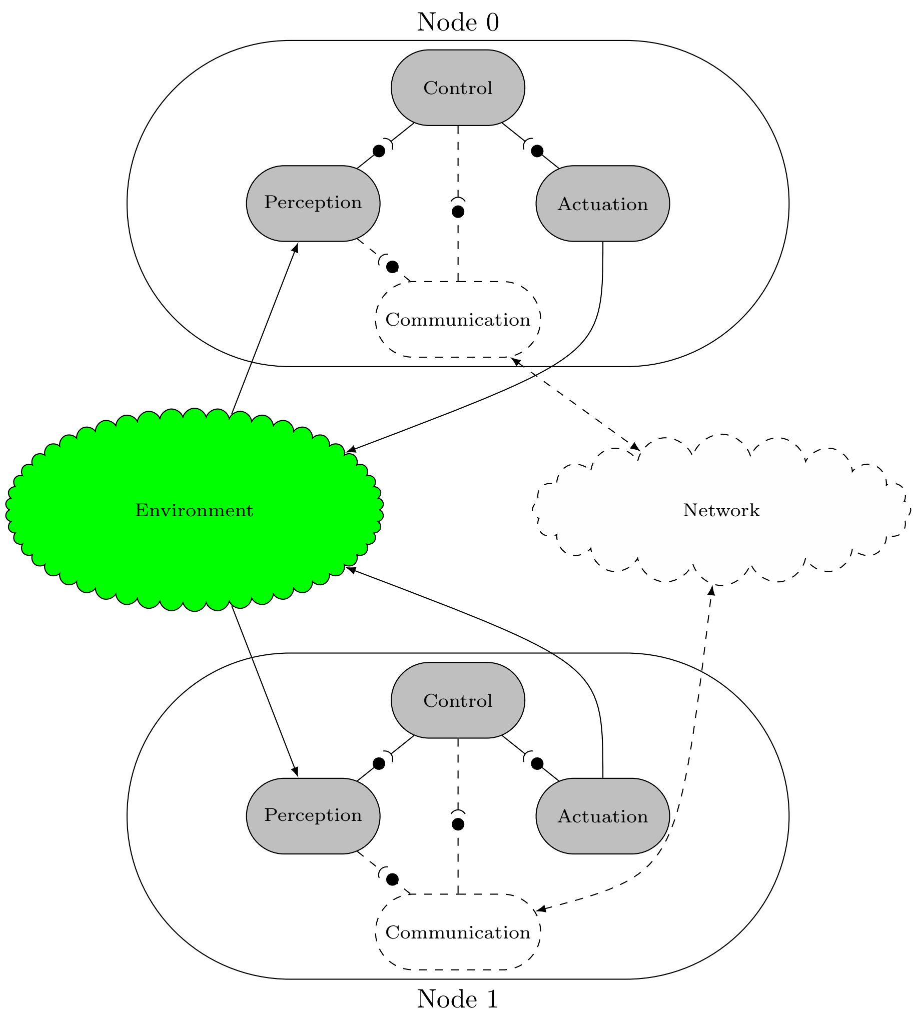


Figure 2.3.: Schematic overview of a CPS. The dashed components represent optional components that disambiguate a CPS from a basic embedded system.

information describing the phenomenon that was observed. The result of this process is *Sensory Information* that may be used by any other CPS component to react to changes of the physical phenomenon in the environment. In the following, the most important terms are defined:

Phenomenon denotes a measurable physical property of the environment or an object within it.

Sensor is a physical component transforming a physical phenomenon such as pressure or speed to an electrical signal.

Sensor Data denotes the raw digital output of a *Sensor*.

Sensor Information denotes the enhanced information that is formed by extending the *Sensor Data* with *Context* information to unambiguously describe the phenomenon.

Smart Sensor is a component which outputs sensor data on one or more phenomena including additional data regarding the context of data acquisition such as quality, time or position. The data is delivered through a digital communication interface. The *Smart Sensor* is composed of at least a single *Sensor*, a communication interface and a processing unit handling the necessary signal processing.

Such a sensor is a component observing a physical phenomenon and transforming its state into an electrical or digital representation. In CPS, sensors are used in the *Smart Connection Layer*, the *Data-To-Information Layer* and the *Cyber Layer*. In the *Smart Connection Layer* they acquire and distribute data on phenomena observed in the environment. In the *Data-To-Information Layer* the CPS sensor attaches context information to the raw sensor data, which are used in the *Cyber Layer* to enable fusion of sensor information to enhance quality. The functionality of the different layers need not be contained in a single node. Consequently, sensors used in CPS need to provide data on attributes defining their behavior such as: interface, modality, context and quality to enable the CPS to use them dynamically.

Interface may either be analogue through an electrical voltage or current or digital through a field bus interface. The analogue interface is very easy to use, but does not provide any meta information besides the raw measurements. The digital interface may provide health information or parameters of the sensor, but a suitable field bus interface is needed to access the sensor data.

Modality describes the type of phenomenon the sensor observes. Generally, these relate to physical parameters like temperature, pressure, humidity etc. However, there exist other sensors such as object detection sensors, which do not directly relate to physics. Alternative sensors based on cameras might even detect special features such as speed limits for parts of the road or the gestures of humans.

Quality describes the meta-information provided by the sensor regarding error-probability, uncertainty, latency and the availability and quality of context information such as position and time.

Smart Sensors enhance the functionality of a sensor with the ability to deliver more contextual data. In general, it is able to self-test its functionality to output contextual data regarding the quality of the output data on run-time. Additionally,it may contain more functionalities such as dynamic estimation of its current position and time synchronization to provide self-sufficient sensor information. A *Smart Sensor* producing sensor data and the necessary contextual data is an information producing component in the CPS, which may be added and removed dynamically, because the system can dynamically assess the impact of this structural change and adapt to it. In [172] Zug describes an architecture to develop smart sensors hierarchical based on different functional groups with a focus on self-testing and quality estimation. In this case the necessary contextual information is already contained in the sensors. However, a general problem of *Smart Sensors* is the acquisition of complete context information. Relevant information such as position, time and network quality parameters can only be obtained by other sensors in the system. Therefore, *Smart Sensors* are typically composed of multiple internal sensors that are combined to form the necessary *Smart Sensor* needed by the CPS.

Some examples of *Smart Sensors* which additionally output context data are the following:

Context-aware Distance Sensor is a digital distance sensor with a known position, orientation and an embedded digital data sheet. It enables the subscribers to transform the acquired distance information into positions in a local map. This enables further automated processing such as the combination of multiple distance sensors to virtual speed information.

Car-Detection Cameras are distributed along roads to account car tolls on roads. They acquire high-quality sensor information on cars on the roads. This information consist of at least an identification of the car, its position, the time of detection and possibly its speed. Multiple of these sensor allow the deduction of average speeds on roads and road usage.

GPS Sensor are inherently contextual sensors, as they provide tempo-spatial information on an object with with a known uncertainty. The uncertainty of the sensor is inherently dynamic, as the provided accuracy of the tempo-spatial information depends on communication links and quality to the available satellites.

Weather Station typically consists of multiple heterogeneous low-quality sensors. They provide information with on multiple modalities such as temperature, humidity, air pressure and illumination. To enable a global the individual weather stations' data is combined which needs tempo-spatial context information. Consequently, the produced data contains at least information on the tempo-spatial context.

2.3.2. Actuation

The *Smart Actuator* is the counterpart to the *Smart Sensor*. While the smart sensors produces self-sufficient information on physical phenomena, the smart actuator consumes such information to physically change the phenomena. It needs the same amount of context to the information to be able to assess the correct phenomena to be manipulated. Typically, content-based routing mechanism may be used to transmit information on relevant phenomena to the correct smart actuator. The information on quality may be used by the *Smart Actuator* to check the manipulation for validity to prevent possible dangerous operations. An example is an emergency breaking system, which produces a false-positive emergency information. If correct quality information is available, the smart actuator can decide not to execute the break command, because the necessary validity of data is not guaranteed. An even more sophisticated CPS might prepare an emergency breaking, but waits for additional information to take a more reliable decision.

In general, *Smart Actuators* are combinations of sensors, controllers and actuators being able to check the actual actuation whether it is executed or not. In the case of an internal error the *Smart Actuators* are able to diagnose the error and feed back the information to the other components of the system. They follow the same classification as the *Smart Sensors* regarding Interface, Modality and Quality. Some examples of actuators in CPS are:

Motors allow rotational or lateral movement of parts. A typical example are the breaks of a car that create a negative acceleration. However, these actuators do not need to be smart as they are designed to supply their effect with very high reliability as stated by the ASIL D of the ISO26262.

Piezo Elements are another type of actuator which are able to create low-amplitude lateral movement. This may be used to create sounds, heat or vibrations. This actuator is very unreliable and therefore used for tasks where no reliability is needed. It can be extended to *Smart Actuators* by using additional acceleration sensors or microphones.

Electrical Heaters are used inside a control loop, as the heating is typically used to control the temperature of an element. An example is the seat heating of cars, which increases the temperature of the seat untill a user specified temperature is reached. Since these actuators are only used inside of control loops, the actuator, the sensors and the control loop form a *Smart Actuator*.

Leds are used to create light. This may either be used to indicate system states to users or to illuminate the environment. In the case of state indication they are often combined with a self-diagnose test, that allows the user to access the functionality of the actor. This mechanism is not a *Smart Actuator*, because human interaction is necessary to evaluate the quality.

2.3.3. Processing

Processing of sensor information in the CPS is done by distributed processing units. These units consist of a Central Processing Unit (CPU) or MCU and a software abstraction of the hardware, which typically consists of an Operating System (OS) or a *Hardware Abstraction Layer*. The processing components do not influence the environment. They take various sensor information as input as well as a processing specification and output the processed information. However, the processing done can be of arbitrary complexity. The computational power regarding CPU and available Random Access Memory (RAM) vary heavily depending on the envisioned application. A low-power WSN processing unit uses a MCU such as Atmel AVRs with less than 100kB of RAM. Whereas, a robotic control Personal Computer (PC) typically uses general purpose CPUs with billions of operations per second and multiple Gbs of RAM. As a consequence, the abstraction used in the different systems are adapted to the capabilities of the system. Therefore, a low-power system is more limited in the possible processing it can execute. Additionally, pc-like general purpose systems provide the ability of executing interpreted programming languages, which allows the distribution of programs in a uniform way without caring for the actual hardware. In contrast to this flexibility, MCU-based system often provide better support for real-time and reliable execution of the functionality. Therefore, critical static functionality of a CPS is typically implemented as fixed-function units on MCUs. In very high-reliability cases the hard and software of these units may even be fault-aware and enable the detection and possible correction of computational faults. Reliability and real-time may be incorporated in the functionality as QoS parameters in the form of latency and validity. The following section describes classifying parameters of processing units found in CPS:

Computational Power denotes the amount of computations a processing unit can execute per time frame. More complex tasks need more time when the computation power is low.

Memory denotes the amount of non-persistant storage the processing units has for historical values and temporary data.

Flexibility denotes the ability of the processing unit to change the executed functionality by either reconfiguring existing programs or by being updated with new programs.

Reliability denotes the error probability of the executed calculations

Real-Time Capability denotes the probability of the executed functionality to provide the result within a specified time margin. There are two flavors named hard and soft real-time, where the first one specifies the probability of lateness to be zero.

2.3.4. Communication

To connect *Sensor Actuators* and *Processors* in a CPS, one or more communication links are needed. These links acquire information from the individual component and distribute it to the different components. There are multiple low-level protocols to network the individual components depending on the application's requirements. The individual solutions can be classified based on their expected message timing, their supported topologies, their timeliness of delivery and their reliability. The different classes are presented very shortly in the following:

Expected Message Timing describes the periodicity of the messages generated by the applications. Periodic communication systems expect the application to deliver messages in a fixed period, whereas, sporadic communication systems allow the delivery of messages at any point in time. These sporadic messages typically denote events changing the state of a component or the environment.

Supported Topologies describe the type of connectivity layout the communication links support. This ranges from single-hop communication in bus-like structure over single-hop broadcast communication to multi-hop routing using *Ad hoc On-Demand Distance Vector (AODV) Routing* [121].

Timeliness of Delivery describes the existence of real-time guarantees for certain messages transmitted in the network. Typically, purely periodic communication systems allow the computation of message transmission tables offline and therefore guarantee timely delivery of any message, as provided by the *Time Triggered Protocol (TTP)* [89]. Other communication systems may only provide real-time guarantees for specially designed messages, as provided by *Controller Area Network (CAN)* [1]. Some communication systems provide no guarantee on timing of delivery at all such as *Ethernet* and *Wireless LAN*.

Reliability denotes the ability of the network to detect delivery faults and mitigate them. This ranges from automatic retransmission of lost messages over *Forward Error Correction* to prevent the loss of messages to fully redundant systems that transmit messages in parallel over multiple separate connections.

All these parameters highly influence the provided QoS of the communication. Depending on the actual task of the CPS it is necessary to supply the network with appropriate QoS specifications to guarantee the functionality of the individual components. The guarantee of the specified QoS parameters needs to be provided by the underlying network. However, it is possible for the network to not be able to guarantee the parameters. In this case the CPS needs to adapt itself or disable the functionality. A typical case for the necessity of QoS parameters are distributed control loops. They are typically defined in periodic manner and need information to be delivered with at least the same period as the control loop itself. Any delay in the delivery can jeopardize the stability of the control.

In the case of broadcast communication, no addressing is necessary to link individual components of a CPS. However, broadcast communication does not scale well and therefore, for larger CPS some kind of information routing is necessary. A classical approach is routing based on identifiers of components, which designate source and destination address. However, this is inflexible, as it provides no information on the actual data the components need to share. For statically designed systems the designer may hard wire the components' links based on the necessary information, but for dynamically composed systems some kind of content-based routing needs to be applied. In the case of CPS one possible source of the content-based routing information may be the context data. In this case, the individual messages or events need to contain the context additionally to the data. Two types of communication abstraction are well suited for content-based routing in dynamically composed systems. They are described in the following:

Publish/Subscribe (P/S)-based systems uses a communication paradigm proposes a means to communicate events without space, time and flow-coupling [63]. This enables complex large-scale dynamic networks to consist of autonomous units, which communicate and act independently. The focus of these systems is on the data that is communicated rather than communication itself. Compared to traditional communication paradigms like client-server or multicast datagrams, P/S provides content-based dissemination of data in a scalable way, as visible in Figure 2.4. Over time, different flavors of P/S systems such as topic-based, content-based and hybrid systems emerged. The topic-based systems attach a tag to each event, which is called topic, and allows to filter data based on this topic. The content-based systems apply filters directly to the content of the events. The hybrid systems combine both mechanisms by attaching topics to events and allow filtering on known attributes of events.

Service Oriented Architecture (SOA) use machine-readable service descriptions together with a lookup system to automatically connect individual components. The lookup of services is executed by a possibly distributed service repository. This containing the necessary information on the description of the service and its current location in the system. In contrast to P/S-based systems the service scheme does not decouple the flow of client and server. Typically, the client waits until the server has processed the request and delivered the answer. This allows for an automatic synchronization between client and server, but also creates a stronger coupling between the components. Another important aspect is the distribution of the service repository. A fully distributed service repository creates a very scalable dynamic system, but also has large overhead to keep the data up to date or to query the repository. On the other hand, a centralized repository is easy to update and search, but is a bottleneck for the growth of the system.

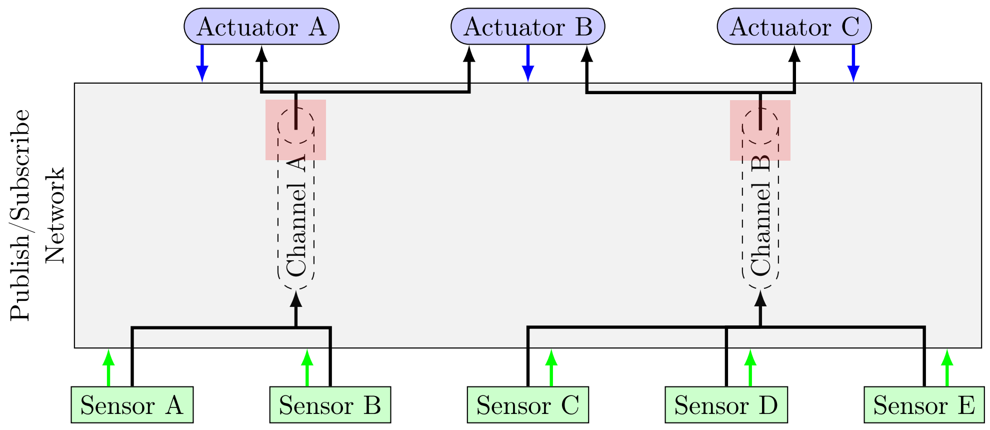


Figure 2.4.: General structure of a P/S middleware connecting sensors, actuators and controllers of CPS.

2.3.5. Storage and User Interface

In addition to the communication of data, some CPS need to store data for later use. This storage may either be temporary or permanent depending on the indented usage. However, only some CPS provide storage facilities to keep data permanently. Storage is also provided by the memory of processing units, see Section 2.3.3, but this storage is mostly short term and buffers data based on time, size or other specific requirements. The short term storage is mostly handled internally by the CPS, whereas, long term storage needs to be activated explicitly in the CPS. Systems such as Aurora enable the user to activate storage components to persistently store data for later queries.

In general, long term storage needs to be explicitly activated by the user. The user also needs to actively request the contained data. Consequently, permanent storage and later delivery can be considered part of the User Interface (UI) of the CPS. The UI in general consists of all components that enable an interaction between the user of the CPS and the system itself. Typically, the UI consists either of an API to query information on the system using the communication facilities, see Section 2.2.1, or of display and input facilities for direct interaction. An example are modern cars, which have multiple buttons and displays that enable the user to change the behavior of the system in various ways, like dynamic adaption of steering aid strength or change of the ride's height. On the other hand, WSN have no direct UI components, but may be monitored and configured using the communication links used for data dissemination. The power of the UI components can range from a purely monitoring functionality as in the WSN forest fire detection scenario to a fully reconfigurable system as provided by an intelligent robot in the dynamic navigation scenario.

The storage components of a CPS can be classified based on the time horizon of storage and the existence of a long-term storage option. The UI components can be classified based on the power of the monitoring and configuration components.

2.4. Summary

The Sections 2.2.1 - 2.2.5 describe the functional and physical components of a generic CPS. The description shows that CPS are very complex systems consisting of multiple components in hardware and software and adhering to multiple functional and non-functional requirements. The individual development of each component is already challenging, but the integration of all these components without violating requirements and still guaranteeing functionality is a very complex and error-prone task, as described by Bangemann et al. [34].

The complexity of the system can only be reduced by either dividing the system into smaller parts or by using high level abstraction layers. The first approach relates to the traditional software engineering goal of reusage of components. The second approach relates to the software engineering goal of information hiding, which encapsulates details of functionality and enables separation of the development within a component vertically. However, general abstractions are not applicable to CPS, because CPS need to be adaptive regarding their environment and their structure. This can only be achieved by providing a maximum of information to all the functional layers of the CPS. An example are real-time requirements, which are very difficult to be included in flexible abstractions. The current development process solves this problem by separating the scheduling and the timing constraints from the used programming abstraction, which induces additional challenges during integration of separately developed components. Edward Lee [98] proposed a change in the development process by natively include time and timing constraints in the abstractions. This allows combines the benefits of the independent development of traditional abstraction without the additional integration effort. However, the abstractions need to gain enhanced functionality to handle violations of constraints on run-time. If this approach is generalized to the whole context of the CPS, it enables a system composed of basic components on run-time. Additionally, the system may even adapt on run-time because all design decisions can be implemented in the system providing additional adaptivity. The Karyon [52] project aimed to provide an architecture to support reliable adaptive systems based on individual components.

The key to enable this new design and development process for the development of CPS is the definition of the context of the system and the inclusion of the context in the abstractions used to develop and integrate the components of the system. Consequently, all functional components need to be context adaptive, since the context of their supplied information may change at run-time. The first and most important step towards this goal is the development of a generic architecture to handle and distribute the sensor data and context in and between the components of the system. The aim should be to have as little as possible manual developer and designer interaction with the system in the configuration phase to allow the system to adapt to the environment. The challenge is to still fullfil the applications' requirements on information quality and context quality, which also enables the statement of real-time guarantees.

The context contained in the abstractions can be viewed as a way to decouple the components. In strongly coupled system the engineer hard-codes the contextual information during design-time. Consequently, the components of the system cannot be separated and recombined unless the contextual assessment of the information shared between the components fit. An example is a car using two radar-based distance sensors in the front to enable reliable ACC functionality in combination with the engine control unit. If one of the sensors is moved to the back of the car, the fusion of the data does not produce correct output anymore and the resulting functionality will be erroneous. This is caused by the change of context of the produced information of the back sensor to something, which does not fit the hard coded context in the engine control unit.

Therefore, the automatic composition of the system is hindered when components are strongly coupled because the context of their abstraction is not extensive enough. The flexibility provided by the explicit handling of context of a component directly relates to its capability to be reused and to be reconfigured on run-time. The following paragraph provides a general overview of the couplings between components limiting the flexibility of the components. Some of them (Time, Space, Flow) directly relate to the couplings described by Eugster et al. [63] for P/S-Systems. Others such as (Data Format, Semantics and Context) are described by Sheth [145]. All these couplings may be present in CPS, especially in the *Smart Connection Layer* (Section 2.2.1) and in the *Data-to-Information Conversion Layer* (2.2.2)

Time coupled systems require the components exchanging information to be active at the same time. Consequently, a system uncoupled in time needs the ability to store events on route to deliver them on reactivation of the participating node.

Space coupled systems need to address the individual components explicitly. Systems uncoupled in space do not need any address to establish a communication channel between components. Of course, other means to identify the endpoints of the communication or the communication itself are necessary. One example of such alternative specifications is content-based communication, where communication partners state their interest in data instead of participants of the communication.

Flow coupled systems enforce a single flow of control between the participating communication partners. This is typical for client-server systems, in which the client waits for the reply of the server. Systems uncoupled in flow enable their components to keep their autonomy even during communication.

Data Format coupled systems need the components to know or agree on a certain data structure for each communicated event during the design-phase of the system. Systems uncoupling their components against the data structure of the communication either need a generic description languages or use meta-languages to describe the data structure. *XML* is an example of the first, whereas *IDL* is an example of the later.

Semantics coupled systems need to agree on the semantics of the communicated data in the system's design phase. A system decoupling the components semantically needs an ontology to translate between the different vocabularies and properties of the terms of the individual components on run-time.

Context coupled systems specify a static context for different components on design-time, which consists of tempo-spatial information, reference points and uncertainties of different observed physical phenomena.

Currently, two possible solutions exists to solve the generic challenge of dynamic composition of CPS. The first uses a powerful domain specific language to enable flexible acquisition of data for each component. Dynamic modification of these statements allows the system to adapt to failures of other components or to changes in the environment. The other approach uses components with special interfaces to enable automatic adaptation based on the context description. The first case moves the adaptation mechanism to the application, whereas the second approach handles adaptation internally. In both cases a component exists, that executes the acquisition commands or links the individual components. In general, this component is distributed in the network, but some systems may implement it centralized. Depending on the mechanism existing systems exhibit strong differences in capabilities regarding sensor data processing and filtering. Additionally, the systems differ heavily in context description and the statement of requirements regarding data, context and quality. Finally, scalability, resource efficiency and computational complexity are different, too.

The next chapter investigates the capabilities and limitations of different existing systems aiming to solve the task of dynamic composition of CPS and compares them. The final goal is the assessment of existing solutions regarding the stated challenges and the inference of the challenges this thesis needs to tackle.

3. State of the Art

This chapter describes existing work tackling the challenges of CPS. In the beginning, comparison criteria are described based on the challenges identified in Chapter 2. Afterwards, the individual work is described and evaluated using these criteria. The chapter ends with a conclusion regarding the current state of the art of dynamic composition of CPS.

3.1. Comparison Criteria

To evaluate and compare existing work regarding dynamic composition of CPS, a definition of criteria is necessary. These need to reflect the challenges as described in Chapter 2. The criteria are categorized in five groups which are describe in the following sections.

3.1.1. Couplings

As stated in Section 2.4, couplings between components of CPS hinders independent development. The more couplings exist between the individual components, the more coordination is necessary between the developers of the components. To enable a dynamic composable and distributed CPS, it is beneficial to limit the coupling to the minimum. The described coupling forms a good comparison metric to rate the level of decoupling of a CPS abstraction system. The relevant couplings are the following: Time, Space, Flow, Data Format, Semantics.

3.1.2. Context

Section 2.1.3 describes the importance of the context of data exchanged in CPSs. Additionally, a minimal set of context attributes is specified to enable later reasoning and processing of incoming information. This set is used as the context criterion to rate CPS systems regarding their ability to handle, preserve and process contextual information. The set consists of the following attributes: Time, Position, Uncertainty and Semantics.

Semantics is also part of the couplings between components, because explicit statement of the semantics does not generally decouples the information processing components semantically. It is still possible that the used communication system and the information processing does not allow exploitation of this contextual attribute.

3.1.3. Quality

Quality is a special context attribute as it has two important sub-aspects. The quality of the exchanged data together with the latency is considered QoS of the sensor and the distribution network, whereas, the QoC describes the completeness and accuracy of the contextual data.

3.1.4. Distribution Type

CPS are inherently distributed systems. As described in Section 2.1, they consist of multiple autonomous networked components. For these components to work together, the used distribution mechanism may either rely on a central node or distribute computation and decision making over the components. A fully distributed system enables maximum scalability and robustness, since nodes are mostly independent of each other. Additionally, such systems often employ failure mitigation strategies handling the crash of individual nodes providing additional robustness. This flexibility is expensive. Fully distributed systems need a lot of network bandwidth and time to reach consensus. Centralized systems are cheaper, but at the same time not a robust since a failure of the central node disables the whole system. A third class of system uses a master node to simplify consensus and organization on a distributed system. This master node may either be defined on design-time or be chosen on run-time from the amount of existing nodes.

This comparison criteria classifies the systems into one of these three types: centralized, distributed or master-based.

3.1.5. Filter Model

Each component of a CPS needs sensory data to be able to perform its duty. To enable a dynamic coupling, the necessary data paths cannot be built statically. An approach towards dynamic binding is the specification of necessary parameters of the sensory data. Consequently, the task of delivering matching data is delegated to the underlying middleware. To this end, filter expressions may be used that select data based on specific properties or values of contained attributes as described in Section 2.2.3. To specify such filters, multiple approaches are possible. The database community uses Structured Query Language (SQL)-like languages to specify them. SQL-like languages are composed of expressions containing *SELECT, FROM* and *WHERE* statements. These statements allow the specification of interesting attributes, sources and ranges. P/S systems typically embed filters in the subscription specification. These filters work on topics, types or individual attributes of the data depending on the P/S system used. Another approach towards filtering of data is based on patterns. It uses pattern descriptions such as regular expressions or query-by-example as input and generates appropriate filtering rules based on them. The last method to filter data is based on the specification of processes, which only propagate data when the data fulfills the filter. These processes are typically not specified using a specific language, but are programmed in a native programming language and selected as part of the channel specification. Consequently, the filter model is either SQL-based, Subscription-based, Pattern-based or Process-based.

3.1.6. Filter Expressiveness

The expressiveness of the filter specification can be measured for different filter operations. These operations form three groups:

Elementary Filters are filters, that check individual attributes of the data. Typical operations for these filters are $<, >, ==$.

Filter Compositions are operations that combine elementary filters to form more complex filters. Typical operations are logical operators: and, or, etc..

Filter Sequences enable compositions over multiple separate data sets. Typically, they are used in the temporal domain and contain operations such as: *BEFORE*, *AFTER*, *CONCURRENT*.

The expressiveness is measured in a scale ranging from $--$, describing no filter expression exist, to $++$, allowing the description of any arbitrary filter.

3.1.7. Processing Model

The processing model describes how specified processing operations are applied to input data to create output data. The following classification is based on the described sensor information processing and reasoning mechanism of Sections 2.2.3 and 2.2.3. This comparison criteria can take on three major forms:

Inference Algorithm (InfAlg) uses existing data as facts in a knowledge base. The processing and filter operations are transformed to rules on this knowledge base. Finally, the inference algorithm is run on this extended knowledge base and puts each inferred data back in the knowledge base.

Relational Algebra (RelAlg) handles data in the form of tuples in a relational database. The specified processing and filter operations are transformed to relational algebra and applied to the database. Systems using the windowed version allow the specification of an additional time or tuple count filter to limit the pairing of possible input data in a Windowed Relational Algebra (WRelAlg). This is often necessary to increase the performance of the system. Additionally, the window specification can be used to limit the memory consumption of the underlying system. Depending on the specific system, data are either consumed whenever they are part of processing operations or not. This has a strong impact on the output of the system, especially when multiple processing operations can be applied to the input data.

Automata systems transform filter and transformation expressions to formal automata. The automata are then fed with the data. The amount of storage needed for each automata depends on the expression creating it. The expressiveness depends on the type of automata used. One major benefit of these systems is the ability to prove the expressiveness and the correctness of the used automata.

Process -based computations of input data use implemented and maybe configurable processes. These cane either be implemented as modules or be described in a special Domain Specific Language (DSL). The processing may include historical data using local memory to enable fusion or aggregation. The computational process output data only, whenever input data is received, but they may output multiple data sets depending on the consumption policy.

Rule computation uses a rule syntax to specify the transformation from incoming data to outgoing data. The selection of the appropriate rules for applied to the incoming data is done automatically by the system without an explicit specification by the user. In contrast to the Inference Algorithm (InfAlg) computations, the Rule selection is only applied on channel creation and not for each individual data set.

3.1.8. Processing Expressiveness

The processing of input data to new outputs can take on multiple forms. Three major groups of operations are used:

Transformation take a single input dataset and apply operations to it to create a single output. Examples are unit conversions or transformations of one physical phenomenon to another.

Aggregation takes multiple input dataset and combines them to a single output using special aggregation operations like *min, max, average* and *median*. These operations generalize the information contained in the input data to extract trends or minimize network bandwidth at the cost of additional uncertainty.

Sensor Fusion can take on three forms (cooperative, competitive and complementary) as described by Elmenreich [61]. Each form needs different mathematical operations, typically in the form of sensor signal processing such as digital filters (*Finite Impulse Response* Filters, *Infinite Impulse Response* Filter or Kalman-Filters).

The expressiveness of each system is graded using a scale from $--$, stating no expressions of this type are available, to $++$ enabling all possible expressions.

3.1.9. Resource Efficiency

The components of a CPS differ in available computation, memory and communication resources. The used hardware range from small MCUs with low-bandwidth communication links such as MicaZ motes used in WSN to industrial PCs used in industrial automation. Based on this heterogeneity of available processing resources, it is necessary to evaluate the general consumption of resources of the individual solutions. This evaluation always depends on the individual implementation, therefore, systems with hight resource consumption are not generally unsuited for the task. However, a system evaluated on an embedded system with fewer resources is guaranteed to be able to be distributed to even the low-power components. There are three different resources, which need to be evaluated: the necessary CPU power, the needed memory footprint and the necessary network bandwidth of the system. These three sub-criteria are based on the description of the processing components of CPS in Section 2.3.3 and communication components in Section 2.3.4. Since a general statement can only be very coarse, a 5 level assessment system is used in the remains of this chapter. The assessment ranges from $--$ over *o* to$++$, where $--$ means high resource consumption, *o* meaning average resource consumption and $++$ indicating very low resource consumption.

3.2. Sensor Description Framework (SDF)

The first category of systems aiming to handle the challenges of dynamically composed CPS are SDFs. These frameworks describe the individual sensors, processing nodes and communication facilities used to couple sensors and applications in a generic way. Consequently, these enable the analysis of the data path regarding attributes such as uncertainty, reliability and latency. Additionally, they may ease the integration of sensors in existing systems on design- or run-time. To achieve their goals, they focus on machine-readable descriptions of sensor characteristics, configurations and data formats to automatically adapt components in the resulting system. The generic computational model of these systems is a linear process model with a possibly flexible amount of inputs and outputs. In the following, the relevant SDFs are described:

IEEE 1451/21451 is a standard originally developed between 1998 to 2012. It considers sensor as Plug'n'Play elements, which communicate with *Network Capable Application Processors (NCAP)* [6]. By integrating electronic transducer data sheets [5] in the sensors, the processors can adapt their behavior to integrate them. To enable this, the communication links need to be specified in a way that IEEE 1451/21451-compatible sensors and NCAPs may communicate without any additional configuration. Therefore, different wired [7] and wireless [2, 8] communication interfaces are specified.

The standard focuses on the low-level description and integration of sensors through description of data types and communication details. The described processing only contains basic signal pre-processing without any filters or fusion mechanisms. Additionally, a discovery mechanism to detect and integrate newly attached sensors is included in the standard. Because of the low level of abstraction, the implementation is very light-weight and resource consumption is low. The Transducer Electronic Data Sheets are binary encoded and thus small and easy to transmit even on low-bandwidth networks.

The used processing is based on the process model with a dynamic amount of inputs and a fixed amount of outputs. This model handles transformations very well, but the amount of existing aggregation and data fusion processes are limited. Additionally, the processes may be used as filters based on the sensory data providing elementary comparisons and logical connectors. Sequences are not filterable by the processes. Processing is directly handled by the (NCAP) enabling a fully distributed system. The systems are loosely coupled since (NCAP) and sensors only need to be active at the same time. Because of the used discovery mechanism, no addresses of sensors need to be known and the data structures are defined for each sensor modality. However, the semantics of the data is only grasped by a fixed amount of sensor types defined in the standard, which is not adequate to completely capture semantics.

SensorML is an SDF originally created by the Open Geospatial Consortium between 2007 to 2014 [117]. It aims to enable fully automated world-wide access to any sensor from any point of the world. To this end, sensor and all processing components are modelled as eXtensible Markup Language (XML)-described web-services. The sensors' description contains the relevant context like position, orientation (if applicable), measurement uncertainty and more. Accessing sensors is done by communicating with a web-service through well established service frameworks such as SOAP. The processing is also specified using XML as linear multi-input multi-output processes. The mathematical computation of the individual process is described using MathML [31]. The service discovery necessary to find a specific sensor is not covered in SensorML and needs to be handled externally. The completeness of the information processing chain from data acquisition over transport to processing enables the usage of described chains for modelling and simulation as well as configuring and using enabled sensors.

The extensive description of the sensor and processing services decouple the sensors and processing services regarding data structure and quality of the sensors' information. The lack of an embedded sensor discovery mechanism creates a space-coupling between the sensors and the consuming services. Depending on the individual content of the description and external discovery mechanism a semantic-decoupling is possible. The usage of web services and their client-server communication architecture enforces a flow-coupling. The resulting systems are generally time, position and uncertainty-aware, since these are mandatory attributes of sensors. Reliability and bandwidth are not considered in the descriptions at all.

The used SOA creates a rather large overhead concerning memory. The used XML-based description creates a large network bandwidth consumption. The resulting systems are generally not suited for embedded applications. Currently, only a single *Java*-based implementation [162] of the standard exists, which is not usable on embedded systems.

The processing model is process-based and enables mathematical operations on input data through MathML. This enables arbitrary transformations, but aggregation and data fusion need to be implemented manually and may be configured using XML. Filtering may also be implemented using the process model.

MOSAIC by Zug 2011 [172] is an SDF focusing on the detection and handling of possible failures of sensor processing chains. It uses sensor data sheets similar to the *Transducer Electronic Data Sheets* of IEEE1451, but extends them with information regarding failure propabilities and failure effects. It contains two independent sensor processing chain development frameworks using Mathworks Matlab and Python as basis. The user is able to specify a data path containing input sensors based on modality together with sensor data selection components and fusion mechanisms. The input data is acquired using a P/S-based communication system and each specified node is considered an instance of the sentient object model, see Section 3.6.

MOSAIC enables fully failure-aware acquisition and processing of sensor data. Uncertainty is mapped to failures and not handled separately. The used P/S system reduces the necessary couplings to acquire sensory data to time and semantics. The semantic coupling is induced by the need for pre-known sensor topics necessary to establish the channels between the sentient objects. The awareness of the system is limited to failures, position and time since uncertainty is considered to be one type of failure. The resource consumption of the created systems depend heavily on the used backend. The *Python* backend is not able to generate embedded solutions because of large memory and CPU overhead. The *Matlab* backend can generate embedded code with a slight overhead in memory consumption because of the *Matlab*-specific scheduler. The bandwidth consumption is generally low, because of the used specially tailored P/S system *FAMOUSO*. The expressiveness of the system is good concerning sensor information because of the extensive data sheets used, but lacks in the processing part as processing needs to be implemented by hand. The system only generates code skeletons for this purpose.

AutomationML AutomationML [50] is a description language for industrial processes and entities in the Industry 4.0 context. It focuses on the description of necessary configurations and connections between entities in an automated factory environment. The standard is very expressive regarding the entities existing in a typical factory. Typical objects such as robots, sensors, actuators and communication infrastructure , see [49], can be described by the language using XML datasheets. However, the main goal of AutomationML is a homogeneous description of a flexible production environment to unify the data exchange between different industrial tools. The language is very high-level and optimized towards factory automatization means, but allows an extension of the described entities towards different scenarios.

AutomationML aims to enable the description of factory configurations that are automatically applied to the factory. In the future, the entities contained in the factory will use the AutomationML descriptions to configure their behavior and adapt to them. The implementation of the entities is unknown by AutomationML. Therefore, the parameters time and flow coupling and the resource consumption cannot be evaluated generally. Space coupling is prevented by the communication description and semantic description of factory elements, which is a native part of the language. However, some semantic coupling may be present, whenever extended entity descriptions are used because AutomationML provides no means to relate different extensions to the basic entity description mechanism. QoS descriptions are supported, but the application and provision of conformance to them is up to the implementing entity.

AutomationML allows the specification of behavior of entities using a Automata approach, see [51]. This enables the description of automata, which react to incoming events and change their state accordingly by executing different actions. The automata guards used to filter the incoming events can be specified as a programmable logic controller program, which provides large flexibility. Therefore, AutomationML allows the specification of arbitrary filters containing elementary and composite comparisons. However, sequence filtering of events is only possible indirectly using special automata. The automata are tailored towards the execution of actions regarding actuators. Therefore, they provide extensive possibilities to transform incoming events. Unfortunately, they lack native means to aggregate and fuse incoming events. In theory these operations are executable using specially constructed automata, but this creates a large overhead in specification and execution. Uncertainty of incoming events or events' attributes is not considered at all.

Criteria		IEEE 1451/21451	SensorML	AutomationML	MOSAIC
Coupling	Time	x	x	()	x
	Space		(x)	()	
	Flow		(x)	()	
	Data Type				
	Semantic	(x)	(x)		x
Context	Time	x	x	x	x
	Space		x	x	x
	Uncertainty	x	x		x
	Semantics	(x)		x	
Quality	Service			x	
	Context	(x)	x		(x)
Distribution	Type	distributed	(distributed)	(distributed)	(distributed)
Resources	CPU	+	-		o
	Memory	+	-		o
	Network	+	-		+
Filter	Model	Process-based	Process-based	Process-based	Subscription-based
	Elementary	o	o	++	+
	Composite	o	o	++	o
	Sequence	−	o	o	−
Processing	Model	Process	Process	Automata	Process
	Transformation	+	++	+	++
	Aggregation	-	o	-	−
	Fusion	o	o	-	−

Table 3.1.: Evaluation of the described Sensor Description Frameworks against the comparison criteria, see Section 3.1.

Summary As visible in Table 3.1, the described SDF provide extensive descriptions of sensor information together with a varying amount of couplings. The awareness of the individual systems depend on the level of abstraction and the focus of the system. SensorML focuses on usage of generic solutions and wide-spread applicability, whereas IEEE 1451 focuses on ease of integration and MOSAIC focuses on reliability and fault-tolerance. Depending on the level of abstraction, the fitness towards embedded solutions is very different. SensorML provides the highest level of abstraction, but it completely unsuited to embedded applications, whereas IEEE 1451 is specially tailored for embedded systems at the cost of less functionality and abstraction. AutomationML provides extensive descriptions for a special use case, but can be also be applied to general sensor processing. The expressiveness of all system is not enough to enable a dynamically coupled CPS since either processing or selection lack.

3.3. Communication Middlewares

The following section describes *Communication Middlewares* aiming to provide awareness against different parameters and to decouple the nodes regarding different attributes. The first type of middleware are P/S middlewares focusing on space, flow and semantic coupling. Afterwards generic WSN middlewares are described that aim to provide a low resource footprint to enable them on embedded hardware used for real-world WSN. The last type are WSN Ontology Frameworks aiming to provide a semantic decoupling between the individual nodes while maintaining a low resource footprint.

3.3.1. Publish/Subscribe Middlewares

A P/S middleware is a possible instance of a communication component of CPS. The details are described in Section 2.3.4. In general, multiple flavours of P/S system exist, but all of them aim to decouple the components in space and flow. Some even provide time decoupling. The major difference is the used event abstraction and the filter expressions usable by the subscriber. In the following relevant P/S systems are described and compared regarding the criteria stated in Section 3.1.

Scalable Internet Event Notification Architecture (Siena) is a scalable purely content-based P/S-system. It features fully distributed event dissemination based on an statically configured overlay network designed by Carzaniga et al [42]. Filters are expressed over attributes of events, which are tuples of attributes. It supports limited aggregation on meta-level by merging subscriptions and announcements.

The communication between publishers and subscribers is handled by brokers. The brokers create an event mediating static overlay network, which eases the routing of event subscriptions. To support different application scenarios, Siena provides three types of overlay networks, which are hierarchical, acyclic peer-to-peer (p2p) and general p2p, to meet different needs of applications. The overlay network consists of servers forming client/server relationships dynamically based on subscriptions and announcements.

Even though Siena is scalable and efficient, it suffers from its static overlay network, which must be separately administrated and is not useful in dynamic network scenarios. The different overlay networks provide a certain amount of adaptability to applications' requirements. Siena exploits already existing internet technology as the basic communication mechanism. It provides a decoupling in space as expected by a P/S system. However, a decoupling in time and flow is not mentioned by the authors. That may be caused by the used forwarding mechanism in the overlay-network and the used client/server communication mechanism. Even though there is no topic or type used to name individual communication channels, the individual applications still need to know the attributes of the events they are interested in, which creates a semantic coupling between publishers and subscribers. The context of each event is not generally specified, except for time. Therefore position, uncertainty cannot are not generally available to applications. Even though the authors mentioned time stamps to be attached to each event, they did not state the used time source. Siena's distribution model is based on different nodes communicating to servers distributed in an overlay network. From a resource perspective, the used server/client structure is computationally expensive, since each node needs to investigate each subscription or publication on forwarding. Additionally, Siena stores all subscriptions and publications in the nodes to minimize the used bandwidth by aggregating them. This minimizes bandwidth, but increases memory consumption. Finally, Siena contains a sophisticated data description language to describe patterns of content to which events are matched. These filters can be combined to create baseline Complex Event Detection (CED), with the absence of sequence operations. Additionally, no processing expressions are included in the language disabling any event processing besides the meta-level aggregation.

PADRES is a fully distributed P/S system developed by Li and Jacobsen [102] in 2005. It uses an overlay network of broker nodes distributing the events from publishing nodes to subscriber nodes. Subscribers may describe content-based filters using the attribute of the events. The individual filter expressions are attribute comparisons using the operators: $<, \leqslant, =, \geqslant, >$. These can be combined using logical operators such as *and* and *or*, excluding *not*. Sequences of events can be detected using either the *sequence* operator to detect two consecutive heterogeneous events or the *repetition* operator for consecutive homogeneous events. The *sequence* operators additionally need a maximum time between the events. The system is implemented on top of the Jess [68] rule engine, by translating all filter operations to rules forwarding the input events. In addition to the RETE-Trees used by Jess, the authors evaluated a predicate counting algorithm, which used much less memory at expense of latency.

PADRES uses an absolute time model and enables time-based subscriptions. The individual content-filters of the subscriptions are decomposed and distributed in the network to minimize network bandwidth and latency. It contains no notion of event processing and is only able to detect sets of events and deliver them. The processing of these events needs to be done within the applications. The main goal of the composite subscription is the minimization of notifications in the subscribers to save network bandwidth and CPU resources. The resulting couplings of the system are semantic and time coupling, since PADRES stores no events and the meaning of the individual event attributes is application-dependent.

FAMily of AutOnomous Sentient Objects (FAMOUSO) is a hybrid P/S middleware for a wide-range of target systems by Schulze [142]. It uses Template Meta-Programming (TMP) techniques to adapt itself to the applications' needs in different configurations. This mechanism allows the usage on extremely low end 8-bit microcontrollers as well as full-fledged PC-systems. Subscribers and publishers can specify requirements and provisions regarding attributes of the events and the current network context such as latency and bandwidth. These are checked on compile-time or run-time dependent on the configured network. Additionally, a compile-time verification system detects misconfigurations and infeasible requirement provision pairs. It is used as the backend of prototypes for the KARYON project [83], which aims to provide a hybrid system enabling dynamically configured CPS providing safety guarantees.

Using the latency and bandwidth checking system, a limited QoS context can be established for events transmitted via FAMOUSO. However, its generic use-case does not enforce any position or uncertainty information for the communicated events. The necessary topic-names used to establish a communication-channel between publisher and subscriber create a semantic coupling between them. FAMOUSO uses an attribute system to pack information in events in flexible manner. This system specifies the attribute structure as a compile-time structure and encodes the type of each attribute together with the used data range in the event. The events are then dynamically unpacked on the subscriber side. Because of limitations in the used *C++* TMP system, only a limited set of primitive data types like integers and booleans are supported as attribute data types. The used filter model is, as expected of a P/S system, Subscription-based. FAMOUSO enables filters to be specified as expression templates on the specified event and channel attributes, providing a very rich elementary filter set. The elementary filters can only be combined using a logical *AND* providing a very limited set of filter composition operations and no filter sequence operations at all. As typical for P/S-systems flow and space coupling are not existent but time coupling is. The resource consumption is extremely low as tests and evaluation on Atmel 8-Bit microcontrollers have shown. However, no processing functions are contained in the middleware.

Robot Operating System (ROS) is a hybrid P/S and service-based communication middleware for robotic applications originally described by Quigley et al. [128]. It uses a central master to manage channel creation and service discovery between the nodes. In ROS a node is program running on a PC machine and communicating either via TCP/IP or UDP/IP. Currently, extensions are available enabling a multi-master ROS, which behaves similar to a P/S-system using an overlay network. Each master acts as a broker in such a system, however, these extension are not stable yet.

The communicated event types and service call messages need to be defined on design-time of the system, but the choice of message for each connection is done on run-time. Filtering and processing of information is done using specialized nodes for concrete application scenarios. Consequently, ROS contains a large set of already defined message filters and processors for different use-cases such as point cloud operations or image processing. However, each filter or processing node is only executed locally, therefore, the performance of these nodes is limited especially in the case of high bandwidth communication. Nevertheless, this large ecosystem makes it a versatile tool to implement, test and evaluate robotic applications.

The centralized master enables a very limited storage of messages, that may be delivered even if one node is currently not active. Therefore, ROS is only partially time-coupled. ROS provides decoupling regarding space and flow as most P/S systems do. Additionally, ROS is a topic-based P/S-system, which creates a semantic coupling since topic names need to be known. However, this is mitigated by a namespace support enabling the creation of generic topic name patterns for special use-cases.

ROS is time-aware since the central master attaches time-stamps to most of the events. Position is typically handled by the *TF* sub-system of ROS enabling the specification and translation between arbitrary Cartesian coordinate systems. Uncertainty, QoS need to be handled by each node in the system and are seldom implemented. The resource consumption of ROS is high for serialization and connection establishing. The memory overhead is very large because of a heavy usage of shared pointers and threads within the general implementation. The network bandwidth is high to establish a channel, since XML-based communication is used in this case. The serialization of events is efficient because of the specialized message system. A message can contain all primitive data types such as integers, floats and booleans. Additionally, it can contain vectors of these values and other messages as members. This enables a very fine-grained message hierarchy. However, this hierarchy needs to be established on design-time, since messages need to be compiled to *C++*-header files and integrated in publisher and subscriber. The processing itself uses a general language approach, processing nodes may either be programmed in *C++* or *Python* and focus on reusability. Most processing nodes can be configured on execution or on run-time enabling a very powerful processing framework, but with a large overhead in the creation and usage of the individual processing operations. Since the overhead of communication between nodes is large, the granularity of the processing operations is large as well. ROS provides high-level processing operations for special use-cases such as point-cloud resampling or convex hull computations. Arbitrary transformations need to be implemented manually by the CPS designer. Selection of incoming data is handled exactly like processing using specialized nodes to filter the data and forwarding only relevant data, which is quite inefficient and seldom used. In consequence, messages in ROS are typically filtered directly by applications.

Summary The described P/S middlewares focus on the decoupling of the components in flow and space, as visible in Table 3.2. This enables an easy creation of autonomous components, which are not directly dependent on each other. Because of the design as communication middlewares, they only provide limited support for storage of events, which typically results in time coupling. Additionally, the filter mechanism depends on content descriptions, which use attribute names defined by the system designer. This creates a semantic coupling between the components of the system. Most of the systems focus on scalability and favor a fully distributed design. The only exception is ROS, which uses a central master in current implementations.

To enable an efficient communication, the systems enable a wide variety of filter operations to be applied to the established channels. This limits the used network bandwidth and also reduces the amount of events needed to be evaluated by the applications. Processing is not very well represented in these systems.

Criteria		Siena	PADRES	FAMOUSO	ROS
Coupling	Time	x	x	x	(x)
	Space				
	Flow	x			
	Data Type				
	Semantic	x	x	x	x
Context	Time	x	x	(x)	x
	Space				
	Uncertainty				
	Semantic				
Quality	Service			(x)	(x)
	Context				
Distribution	Type	distributed	distributed	distributed	(master-based)
Ressources	CPU	−	-	++	-
	Memory	−	-	++	−
	Network	+	o	++	o
Filter	Model	Subscription-based	SQL-based	Subscription-based	Subscription-based
	Elementary	++	++	+	-
	Composite	+	+	o	−
	Sequence	−	+	−	−
Processing	Model	RelAlg	RelAlg	none	Process
	Transformation	−	−	−	+
	Aggregation	+	+	−	+
	Fusion	−	−	−	+

Table 3.2.: Evaluation of the described Communication Middlewares against the comparison criteria, see Section 3.1.

3.3.2. Wireless Sensor Network (WSN) Middlewares

In contrast to generic P/S systems middlewares for WSN are more tailored towards sensor data dissemination and efficiency. WSN are inherently Mobile Ad-Hoc Network (MANET)s, which focus on scalability and robustness. They exploit the large number of nodes they are composed of to provide tolerance against singular sensing errors and node crashes. Another type of networks providing similar functionality are Vehicular Area Network (VANET)s focusing on vehicles as data sources and sinks. The limited ressources of these networks enforcees a specialization in the application domain. Therefore, typical WSN either focus on a specific application type or require the system developer to integrate custom code to implement the application specific behavior. The following section discusses the most common and representing approaches of this domain.

TinyDB is a WSN middleware by Madden et al. [105] published in 2005. It uses TinyOS by Levis [101] on low-power wireless nodes as a basic system. Queries are specified using the SQL-based *ACquisitional Query Processing (ACQP)* language. The queries are distributed into the network by the gateway receiving the query. The distribution is based on a *Semantic Routing Tree (SRT)*, which holds relevant meta information on the routing structure between the nodes and on the sensory equipment of the nodes. The queries are broadcasted into this tree and the results are continuously transmitted by the individual nodes towards the gateway at the root of the tree until the query time is over or a specific stop event is received. The ACQP language enables SQL-like queries, which contain the typical statements *SELECT, FROM, WHERE* and *GROUP-BY* with the same semantics as in general SQL. *FROM* differs slightly, since only a single table *sensors* exist. Additional tables containing historical information may be created by *materialization points*, which are explicit storage descriptions. The language only contains aggregation as processing operation, creating a simple Windowed Relational Algebra (WRelAlg). Aggregations are specified in the *SELECT* block and a special aggregation function. *TinyDB* only contains very basic aggregation functions such as *min, max* and *avg.* However, these can be extended by the user by implementing three functions, that realize the aggregation operation. The distribution of the user-supplied functions needs to be handled by the application using *TinyDB.* The aggregations are distributed in the network and each node executes the parts of the aggregation while the events are transmitted towards the root.

TinyDB enables decoupling in time, because of explicitly specified *materialization points.* The decoupling in space , is realized through the maintained SRT. Decoupling regarding flow is based on the event-based processing of *TinyOS.* Data structure decoupling uses the SQL-based description of the results of the query. Semantic coupling is present, since no predefined attributes besides time exist and the meaning of the individual attributes depends on the sensor. The context of the sensors only contains time. Position is considered to be location-based and is handled by general attribute predicates. A special combination of QoC and QoS is considered through the ability to prioritize events based on the contained information. The authors propose a special *diff* operator measuring the change between two consecutive homogeneous events. The distribution model is master-based, because of the gateway node distributing the query and collecting the events. Resource efficiency is a strong point of *TinyDB*, since it is used on low-power wireless motes with low bandwidth communication. Additionally, the distribution of the computational expensive aggregation conserves CPU power. The explicitly defined storage points limit memory needs and the efficient hierarchical routing saves network bandwidth. However, the maintenances of the SRT also needs memory in the nodes and communication bandwidth. The filter model is SQL-based with a rich set of elementary filters and the typical composition operations. Sequence filters are not available.

EnviroTrack is a WSN middleware by Abdelzaher [16] published in 2004. It focuses on the tracking of activities using the distributed sensing facilities of a WSN. To this end, applications specify events they want to track using the *EnviroTrack* language. The relevant events are filtered using application specific boolean functions. A tracking context is constructed by the system using a specified aggregation function on filtered events. The quality of the tracking context may be described by the application using a freshness value, filtering out old events, and a critical mass, describing a minimum amount of events to be aggregated before a tracking event is emitted. Whenever a tracking context detects an event, a unique object id is created and maintained while the object is tracked. This id and the tracking context, which created it, are delivered to a central repository to enable applications to find the object. Tracking of the object itself is done using an approximate aggregation mitigating the influence of time on the aggregation. The critical mass definition prevents false-positive tracking events to be emitted. Using the language, functions may be attached to the tracked object, which can be called by the application enabling interaction with the tracked objects. A direct delivery of the tracking events is not implemented and every access to tracked object data is done through the Remote Method Invocation (RMI)-Application Progamming Interface (API). Group management is used to distribute the processing of the tracking and the RMI close to the tracked object.

The system is built on top of *TinyOS* by Levis [101]. The application specific tracking contexts and objects are preprocessed and translated to native NesC code. It is implemented on MicaZ motes and showed excellent resource efficiency. The tracking of a tank using magnetometer sensors was possible with the resources of the nodes and a low-bandwidth communication link. *EnviroTrack* enables a decoupling of applications from the low-level sensor in space and time. Flow coupling is present through the usage of RMI. The application needs to know the sensor data to describe the tracking context which couples it to the sensor regarding data structures and sensor semantics. Consequently, semantic coupling is also present. The considered context of the tracked object is time and position. Through the age and critical mass definitions a QoC specification is possible. Filters are specified Subscription-based using application specific predicates and general composition operations such as logical connectors. Sequences are not available. Processing is limited to the aggregation using a Process approach. However, only a limited amount of aggregation functions are available.

PeerTIS is a VANET middleware by Rybicki [138] published in 2009. VANETs focus on the dissemination of data between vehicles. They differ from classical WSN through the inherent high-velocity mobility and consequently highly dynamic topology. PeerTIS proposes a peer-to-peer network between vehicles using internet technologies such as UMTS cell phones and IP-based networks. The authors consider structured overlays to be best suited for the task. Consequently, they adapt the Distributed Hash Table (DHT) idea of Content Addressable

Networks (CAN) of Ratnasamy [130]. Furthermore, PeerTIS uses the geographical and time context of vehicles to optimize the queries and provide a higher quality of service. The position of the cars is stored as a road segment id inferred from the GPS of the car. A DHT stores key-value pairs of data in a hash table by hashing the key and distributing the element to a certain node in the system. The value can be retrieved by hashing the key and finding the node responsible for the hash of the key. PeerTIS directly uses the segment id of the road as the hash of the key. By using this the keys are not randomly distributed to the nodes in the system, but are close the producer. The authors observed that applications on the car are typical interested in information in the local vicinity of the car or in data following a linear set of road segments. In both cases it is beneficial to have the values stored close to the road segment. Consequently, the authors uses geographical routing protocols to find the data. Data which is stored along a consecutive set of road segments need to travel back along the road segments. Consequently, aggregation along the road may be used to save bandwidth. QoS is considered by the authors by equally distributing the storage of key-value pairs on the nodes, because this balances the latency of delivery. However, heavily crowded road segments may need to store more data because more sensors are available. This is mitigated by using the geographical position of the cars to distribute the storage. Densely populated roads will also distribute the key-value pairs more fine-grained. A cache for peers along consecutive road segments increases the responsiveness of the system for queries supplying information for the navigation systems of the cars.

The resulting system is coupled in flow and semantics. The queries issued by a car need to be answered by the respective peer, creating a flow coupling between them. The lack of any semantic annotation enforces the applications to know which keys are present in the system as well as their meaning. The context of the data is time and position, uncertainty and bandwidth are not considered. The system is fully distributed using its structured overlay approach. Selection is Pattern-based using the key as filter element. The processing is Process and limited to aggregation of homogeneous data propagating back to the querying car. The resource consumption is very low for CPU, since no hash computation is needed and the aggregation operations are distributed among the peers. The memory overhead is low, since every node only needs to store a local geographic routing table. The network overhead is also low because of the used aggregation and the inherent locality of the data distribution.

UPSP is a WSN P/S middleware by Tong and Ngai [159] published in 2012. It is a hybrid system with the sensor nodes being based on Contiki by Dunkels, Grönvall and Voigt [59] and Android-based mobile nodes. The system is tailored to a sparse network of static sensor nodes (*SSensors*) deployed in remote locations. *Mobile Brokers* are based on cell phones using Android 1.5. The *Mobile Brokers* are equipped with sensors for position (GPS), time etc, which are called mobile sensors (*MSensors*) by the authors. Additionally, the *Mobile Brokers* connect to neighboring *(*SSensors) using near-field communication such as 802.15.4. Subscribers connect to a central server running a web-enabled RESTfull service and a MySQL database to cache events for offline subscribers. It relays subscriptions to geographically local *Mobile Brokers* to fetch the sensor data. To this end, the *Mobile Brokers* periodically report their position to the central server.

The P/S system is content-based and enables filtering on modality, time and location. However, the composition of these is limited to a single time interval and a single centroid-based location. *Mobile Brokers* may move at an arbitrary speed, but nodes moving faster than 1.5 $\frac{m}{s}$ are excluded from brokerage. Together with the use of reliable unicast mechanism a robust delivery of *SSensor* data is achieved. Unfortunately, the delivery latency is approximately 22 s for *MSensors* and more than 3000 s *SSensors*. The network protocol is very efficient, since *MSensors* use the broadband connection of the cell phones to enable a push-based P/S, whereas, the *SSensors* use pull-based P/S by aggregating the sensory data in memory and delivering them in a single batch to the *Mobile Brokers*. Even though meta-data aggregation is used to enable efficient communication to *Mobile Brokers*, aggregation on the sensory data itself is not considered in the solution. Consequently, the system lack any means of event processing. The resource consumption in the *SSensors* is very low for CPU and network usage. The memory usage is limited to 2 MB, which proved to be enough for the considered example scenario. The *Java*-based *Android* and web-service components use more CPU, memory and network resources compared to the other WSN middlewares.

Global Sensor Network (GSN) The Global Sensor Network (GSN) was developed by Aberer, Hauswirth und Salehi [18, 17] in 2006. It presents a middleware enabling access to *Virtual Sensors* through XML descriptions. The middleware consists of a SQL database to process and query the sensor information together with specialized query processors to handle timing information and multiple wrappers to integrate different WSN hardware. The timing model of the system uses multiple local time stamps that are independently processed and evaluated. Consequently, the ability of the system to filter sequences of events is limited to events originating from the same source. The system provides extensive SQL-based filtering using elementary and composite operations. Sequence filtering is enabled using a window. The XML descriptions contain location descriptions, which are handled together with time data separately by the query processor. Therefore, not all filtering and aggregation operations might be applicable to the whole context. The semantics of the sensor data is contained in the XML description as specification of physical phenomenon of the sensor. The generic architecture is peer-to-peer, but queries are processed in specialized database nodes. These nodes need large CPU and memory resources to handle the sensor event processing and query execution. The individual sensor delivering the data can be low-resource embedded WSN hardware. The centralized database nodes enable persistent storage of the sensor data to decouple the querying application from the sensors in time. The capabilities of the system regarding event processing are limited to simple transformations such as projection of event attribute sets to subsets and joining attribute sets in tuple form. Data fusion operations are not provided. The system handles QoS through active dropping of high-frequency event streams to prevent large latencies in event delivery. QoC and uncertainty of sensor data are not handled by the system.

Summary As visible in Table 3.3, the discussed WSN and VANET middlewares provide a very good decoupling of the composing nodes in space, flow and data type. The systems provide at least position and time context to the applications. However, uncertainty and failure awareness is typically not provided. Quality of context and service are partially provided. *PeerTIS*, *TinyDB* and *GSN* handle QoS, while *EnviroTrack* handles QoC. The resource efficiency is generally very good as the systems are tailored towards low-power embedded devices. *GSN* is an exception because its resource consumption is very heterogeneous. WSN nodes are supported, but the filter and transformation operations are provided by specialized database nodes. The filtering is limited to very basic filters, only TinyDB provides better filtering support at the cost of a higher routing overhead. *GSN* again is an exception as it provides very good filtering. The inherent processing capabilities are limited to aggregation. Data fusion needs to be implemented by hand by the application designer. The semantics of the data are specified and embedded in the application for most of the systems. Only *GSN* provides some semantic abstraction based on physical phenomena observed by sensors.

Criteria		TinyDB	EnviroTrack	PeerTIS	UPSP	GSN
Coupling	Time					
	Space					
	Flow		x	x	x	
	Data Type		x			
	Semantic	x	x	x	x	(x)
Context	Time	x	x	x	x	x
	Space	x	x	x	x	x
	Uncertainty					
	Semantics					(x)
Quality	Service	(x)		x		x
	Context	(x)	x			
Distribution	Type	master-based	master-based	distributed	master-based	master-based
Resources	CPU	++	++	++	o	+
	Memory	+	++	++	o	o
	Network	+	++	++	o	+
Filter	Model	SQL-based	Subscription-based	Pattern-based	Subscription-based	SQL-based
	Elementary	++	o	-	o	++
	Composite	+	+	−	-	++
	Sequence	−	−	−	−	++
Processing	Model	WRelAlg	Process	Process	None	WRelAlg
	Transformation	−	−	−	−	+
	Aggregation	+	+	o	−	+
	Fusion	−	−	−	−	−

Table 3.3.: Evaluation of the described WSN Middlewares against the comparison criteria, see Section 3.1.

3.3.3. Wireless Sensor Network (WSN) Ontology Systems

The problem of semantic coupling embedded in WSN is tackled by WSN Ontology systems. These focus on the specification of the different semantics of the communicated events using ontologies. One application of such a semantic decoupling is the combination of multiple heterogeneous WSN to a single data source. Additionally, it can be used to translate data between independently developed WSN to form a singular homogeneous WSN. The following section discusses some representative systems applying ontologies or semantic meta information to WSNs.

SepSen was developed by Kasi et al. [85] in 2012. It enables a distributed inference of events using a distributed ontology. To this end, the ontology was split manually into different parts, which were deployed to the sensor nodes. On each sensor node a RETE-Algorithm [66] runs and uses the specified ontology to infer events. The goal of the system is to reduce the amount of messages transmitted in the system to save power. On reception of a raw sensor event, semantic information is attached to the event and is forwarded to the distributed RETE-Algorithm. The RETE-Algorithm creates a pattern network using the application defined rules and the knowledge base. Each event passing through a node is either *discarded* if it does not fit any rule, *shared* with another node if the rule needs more than one ontology fragment distributed among the nodes or *forwarded* if inference was possible. The RETE-Algorithm reuses already created events, since those are attached to the knowledge base.

The system decouples the application from the sensory input semantically, since the used ontology enables a semantic matching and transformation between input and output of the nodes. Additionally, the storage of the resulting events in the knowledge base enables a time decoupling as all events are stored. Flow coupling is not explicitly mentioned by the authors. The used serialization mechanism is not explicitly stated either and therefore, data structure coupling cannot be evaluated. Space coupling is not present, since the application states its needed information to the network and the distributed RETE-Algorithm shares the data as needed.

The system proposes a context attached to the events, to enable inference on the contextual information, however, only energy is explicitly stated as context attribute. Quality metrics are not considered in the system. Filtering is based on explicit rules stated by the application. The possible filter expressions are not explicitly stated, but the shown example indicates that typical *Prolog* elementary filters and compositors are available. The processing model follows the Inference Algorithm (InfAlg)model. The processing operations enable aggregation in the form of context annotation.

The RETE-Algorithm is generally computationally expensive and may create an infinite amount of events, depending on the input rules. Additionally, the storage of all inferred events requires a lot of memory. The system has shown a decrease in message count in the example network, but the sharing of events between nodes might also create larger message counts, depending on the distribution of the ontology fragments.

Sensors-as-a-Service (SeenaS) is an event-based service oriented sensor ontology system by Alam et al. [22] published in 2010. It enables the encapsulation of sensors as services with a machine-readable SensorML description, which is used to infer sensor events. It uses ontologies to translate the data provided by different sensors based on SensorML descriptions. The goal of the approach is to translate standardized solutions of the service world to the sensor world. To this end, the authors designed their system with three layers: *Real-World Access Layer*, *Semantic Overlay Layer* and the *Service Virtualization Layer*, which are built on top of an IoT-Cloud. The *Real-World Access Layer* uses adapter to fetch and translate the sensor events from the IoT-Cloud. The *Semantic Overlay Layer* contains different ontologies and policies to enhance the basic sensor events fetched from the IoT-Cloud based on the SensorML description of the IoT-Sensor. Finally, the *Service Virtualization Layer* aims to provide the functionality of the IoT-Cloud in a semantic and service-oriented way.

SeenaS uses *Web Ontology Language* described ontologies with a global vocabulary enabling inference on the individual sensor events. Additionally, access policies are defined to enable role-based access descriptions to enhance the security of the produced sensor data. The access policies are embedded in the rules of the knowledge base used to infer events. Additionally, the prototypical centralized system showed complex event processing by employing complementary fusion to infer sensor events for different modalities from existing ones. Even though aggregation and transformation are not explicitly mentioned, these should be expressible. The filter model uses patterns to match the applications' service needs to existing service-encapsulated sensors. No further filtering is described by the authors.

The system decouples the subscribers of sensor events from the IoT-Cloud publishers in time, space, data structure and semantics. It tries to enhance the raw sensor events by a context of time and position to enable tempo-spatial reasoning on the data. Quality is not considered in any form in the system. The resource consumption is not specified but the used PC indicate a quite high CPU and memory resource consumption. The network bandwidth consumption is not stated by the authors.

Semantic Web Architecture for Sensor Networks (SWASN) is a web-based sensor ontology system by Huang and Javed [80] published in 2008. It aims to translate events between different WSN deployed by different entities with incompatible formats using specially constructed ontologies. It is composed of four layers: *Sensor Network Data Sources*, *Ontology Layer*, *Semantic Web Processing Layer* and the *Application Layer*. The *Sensor Network Data Sources* represent the gateway to different heterogeneous WSN providing sensory data to SWASN. Sensor events fetched from gateways are forwarded to the *Ontology Layer*, which semantically enhances them using the concepts of the ontologies. SWASN follows the hybrid ontology approach, as proposed by Wache et al. [167] in 2001. Therefore, each WSN can use its own local ontology to translate its internal semantics to the global ontology used for inference. The semantically enhanced sensor events are forwarded to the *Semantic Web Processing Layer*, which uses the Jena API [110] to include external reasoners and enable querying of values using *SPARQL Protocol And RDF Query Language (SPARQL)* [127]. Finally, the applications contained in the *Application Layer* can query the information through Hyper-Text-Transfer-Protocol (HTTP). To enable tempo-spatial reasoning the context of the sensor data always contains position and time of the sensor data. The processing of the data uses an inference-based approach on application specified rules.

The system effectively decouples the applications from the WSN in time, space and flow, since all data are fetched, transformed and stored in the centralized SWASN server. The data structure used in the communication with the applications is based on Ressource Description Framework (RDF), whereas, the data structure of the individual WSN needs to be converted by the gateway node. Even though, the context of the sensor data contains time and position, while uncertainty is not considered. The quality of context and service is not in the focus of the system. The filter model uses patterns matched to RDF-triples combined by logical operators without sequence compositors. The processing model uses an Inference Algorithm (InfAlg) approach with a rich set of transformations. Aggregation should be possible even though it is not explicitly stated by the authors. Sensor fusion may be implemented using baseline inference operations, but is also not stated by the authors.

The resource consumption of the system is hard to estimate, since no measurements are published. However, the used RETE-Algorithm is generally computationally expensive. The memory overhead is probably high, because the local ontologies as well as the global ontologies and the whole knowledge base need to be kept in memory to enable the reasoning. The centralized reasoning also needs to forward all the data of the individual WSNs to the central sink increasing bandwidth consumption.

Loosely-coupled Component Infrastructure (LooCI) is a WSN middleware with an included sensor ontology by Hugh et al. [81] published in 2012. The system's main goal is the decoupling of individual components of WSN. To this end, a lightweight component model based on the *Java* programming language is used. The authors used the SUN SPOT WSN nodes with their integrated SQUEAK VM providing a *Java* ME execution environment. The used component model distinguishes between two types of components: *Micro* and *Macrocomponents*. *Macrocomponents* represent a process-like abstraction. They are isolated from each other and provide multiple threads, file system access as well as the inclusion of external libraries. *Microcomponents* are generally single-threaded, may not use external library and are not isolated. The components are bound to each other on run-time and provide introspection and reconfiguration facilities. The communication between the components is based on an abstract event bus. Depending on the binding, events are passed between components based on Inter Isolation RPC (IIRPC) when the components run on the same node, or on network communication based on a *Network Manager*. The events are classified using a type system, providing a topic-based P/S. Each node is connected through the *Network Manager* to other nodes in its neighborhood. It distributes received events of local components using either a reliable or unreliable Unicast or a neighborhood- or broadcast providing a fully distributed system.

The event type system uses an event tree, which also provides an ontology. Events in the tree may be converted automatically from events further down a specific branch to events closer to the root. This behavior is similar to a class inheritance model. A consequence of this event tree ontology is an increasing specificity of the represented event the further down it is situated in the tree.

The event tree system enables efficient type based subscriptions, but no processing of events. To this end, the authors describe an Event-Condition-Action (ECA) based *Policy Framework*. It consists of a *Policy Engine* checking the conditions of the rules, a *Rule Manager* executing the rules in which the conditions are fulfilled, and a *Policy Distribution Component* distributing and integrating new rules on run-time. The conditions of the rules are logically connected attribute comparisons. The actions are limited to the transformation of single events or the execution of component-internal functions. As a consequence, the rule engine enables complex filters and simple transformations, but no aggregation, filtering on event sequences or data fusion is available unless a specialized component provides exactly this functionality.

LooCI components are only coupled in time, since no storage components are directly available. Space, flow and data structure coupling is prevented by the used topic-based P/S system. Semantic coupling is partially mitigated through the ontology event type tree. However, only simple *is-a* relationships can be captured by the tree. The context of the events is not further specified. The system does not consider quality of service or quality of context. The implemented example scenario shows a small memory footprint and a very small CPU overhead for the implementation of the LooCI components. Unfortunately, the authors did not conduct any network usage benchmarks.

Summary Table 3.4 attests all systems a decoupling of components in the semantic domain. For *SepSen*, *Seenas* and *Swasn* it is achieved by the explicit usage of ontologies and inference systems, while *LooCI* follows the approach of a homogeneous vocabulary describing the handled sensory information. The context of the data is only specified in *Seenas* and *Swasn* as time and position. Uncertainty is generally not considered in these systems, as uncertain inference using ontologies is still an open research topic. Based on the used inference systems in *SepSen*, *Seenas* and *Swasn* the resource consumption is rather large, which is often mitigated using application specific optimization such as manual distribution of ontologies. As a consequence, these systems allow many processing operations to be specified in the ontologies and are very flexible in the processing domain. *LooCI* is an exception, as it uses a static vocabulary to describe the sensor events, which needs no explicit processing. Consequently, it has very low resource consumption, but only allows very limited processing. Filtering of the created and inferred events is limited to singular events. Sequences of events are not considered in the composed systems at all.

Criteria		SepSen	Seenas	Swasn	LooCI
Coupling	Time				x
	Space				
	Flow				
	Data Type	o			
	Semantic				
Context	Time		x	x	
	Space		x	x	
	Uncertainty				
	Semantics	x	x	x	x
Quality	Service				
	Context				
Distribution	Type	distributed	centralized	centralized	distributed
Resources	CPU	-	-	(-)	++
	Memory	-	-	(-)	++
	Network	o		(-)	()
Filter	Model	Subscription-based	Pattern-based	Pattern-based	Subscription-based/Pattern-based
	Elementary	+	o	++	+
	Composite	+	−	+	+
	Sequence	−	−	−	−
Processing	Model	InfAlg	InfAlg	InfAlg	Rule
	Transformation	−	(o)	++	+
	Aggregation	+	(o)	(+)	-
	Fusion	−	+	(o)	-

Table 3.4.: Evaluation of the described WSN Ontology Systems against the comparison criteria, see Section 3.1.

3.4. Complex Event Detection Systems

A CED language allows the specification of events as combinations of other events. In general, two types of events are distinguished: Primitive Event (PE) and Complex Event (CE). PEs are low level events that are generated directly by the nodes of the network. CEs in contrast are composed of multiple PEs through application specific relations.

The amount of possible relations between two or more PEs leads to numerous CED systems, which focus on different aspects of the composition process such as time, order or event consumption. Some relevant CED systems are the following:

Snoop(IB) is a CED system by Chakravarthy and Misha [43], which focus on the specification of temporal behavior of event detection. It is developed on top of a centralized DataBase Management System (DBMS) and handles typical database operations as input events.

It contains an event composition language that follows the ECA-principle. Rules executed by snoop are triggered by certain events and a predicate. The events handled by Snoop are classified into *Primitive Events*, which are native to the system and *Composite Events*, which are generated by composite event generation rules. The *Primitive Events* contain typical database operations such as `transaction, access, insert, delete` and *update*. Additionally, explicit user events and temporal events can be handled by the system. The used language to express composite event detection rules resembles SQL. The main difference is the ability to handle the detection of sequences of heterogeneous or homogeneous events. Additionally, multiple events can be combined using logical connectors (*AND, OR, NOT*). The condition part may contain arbitrary predicates on the events' attributes.

Since events may not always relate to a certain point in time, the used language contains the `begin-of` and `end-of` operators to convert time spans into two explicit events. The system transforms the expressed rules to expressions of classical relational algebra to input them in the underlying DBMS. To handle the generation of multiply events, Snoop uses parameter contexts, which are stated by the application, to differentiate how composite events are generated. Snoop supports four special contexts: *Recent*, *Chronicle*, *Continuous* and *Cumulative*. *Recent* only outputs the latest instance of the composite event. *Chronicle* uses an at-most-once consumption policy for the input events, limiting the amount of output events to the number of input events. *Continuous* outputs an event for each *begin-of* an event expression, making it suitable to detect trends in continuous valued data. *Cumulative* includes in each composite event the previous instances of the composite event, creating an aggregation-like behavior.

The memory consumption depends heavily on the used parameter context. Contexts such as *Cumulative* and *Continuous* need memory depending on the amount of input data, whereas, *Recent* only needs a fixed size buffer. The CPU consumption depends on the generated relational algebra operators, but is typically high whenever joins are involved. The network consumption is not addressed by the authors.

The time point model for the individual events is considered as a limitation and in 2003, Adaikkalavan and Chakravarthy [20] extended Snoop to handle uncertainty in the time of events using time intervals for the *Recent* context. The resulting extended system was called SnoopIB, providing a quality of context metric.

The system itself is decoupled in time because of the DBMS storage backend. Space- and flow-coupling is present, since applications need to connect to the central DBMS running Snoop. Semantic coupling is also present, as no semantic information is contained in the events.

SENSID is a CED system especially tailored towards WSN by Kranz [92]. It is based on the Amit CED system by Adi and Etzion [21] and modifies the used language by temporal and spatial matching operators. The system describes events as 4-tuples consisting of type, data, time stamp and position. Complex events are described as predicates on sets of events and are evaluated in *Lifespans*. A *Lifespan* contains all events, that fulfill the predicates and may be contained in the created complex event. A *Lifespan* that actually fulfills all predicates is called a *Situation* and may emit a complex event. Since complex events are only produced when a *Lifespan* ends, the end of a *Lifespan* can be triggered by time or explicit termination events. The situation language, as described by Cardell-Oliver, Reynolds and Kranz [40], already contains predicates to filter events based on elementary attribute comparison and logical *and* connections. Furthermore, the authors added the ability to add new predicates using a plug-in system, providing the user with the ability to implement arbitrary predicates detecting sequences and more complex compositions. All situation detecting nodes work independently of each other and form a distributed system. However, a single *Situation* detection cannot be distributed to different nodes.

SENSID enables the native use of temporal and spatial information for complex event detection and is tested in real WSN on *MicaZ Motes* indicating a very low CPU and memory overhead. However, the described filter predicates are limited to elementary comparison and logical `AND` connections. Sequence filtering may be emulated using time attribute comparisons. The handling of uncertainty or failures is completely up to the application, which needs deep knowledge of the underlying generation processes of these data. The system decouples the applications from the sensors in space and data structure through the underlying P/S-Network. Processing of the resulting complex events is not considered in the language and can only be implemented using the plug-in system.

Abstract Events are a programming model proposed by Katsiri and Mycroft [86]. It extends the classical event definition, which describes state changes of real systems, to abstract events, which describe state changes of virtual systems. The authors defined a *higher-level service* called *Abstract Event Detector (AED)* on top of a P/S system. These detectors take an *Abstract Event* description in *Abstract Event Specification Language (AESL)* as input and use the existing publication and subscription mechanism to provide applications with the resulting *Abstract Events*. The used (AESL) is a subset of *Temporal First Order Logic (TFOL)*. It uses horn-clauses to describe the predicates and the transformation of primitive events to *abstract events*. The (TFOL) rules are executed using *RETE-Trees*, which output an *Abstract Event* on each newly generated event. The generated events are contained in memory and serve as a history, which can be queried. Subscribing applications can additionally attach filter expressions using logically connected predicates on the events' attributes. Sequences of events can only be detected by creating a dedicated (AED).

All (AED)s are managed by a centralized *Abstract Event Repository* creating a master-based architecture. The applications can use the repository to find the responsible (AED) decoupling them in space. The history of the (AED)s enable a time decoupling, because arbitrary past information can be queried. However, older events are not automatically delivered to the applications. The used P/S system can be freely chosen. In consequence coupling between components and applications regarding time, space and data structure are undefined. Since no semantic information is contained in the events, a semantic coupling is always present.

The context of the events contains time and position, but no bandwidth, uncertainty or failures are considered. Quality of context or service is not considered by the authors. The filter model is Subscription-based and enables the expression of complex filters using elementary predicates and logical compositions. However, sequences can only be detected by a dedicated (AED). The processing model uses an Inference Algorithm (InfAlg) approach and contains a rich set of transformations inspired by *Prolog*. Aggregation is not explicitly mentioned, but may be expressed using horn-clauses. Data fusion is not considered either, but may also be expressed in horn-clauses. The resource consumption of the system is quite large regarding CPU and memory because of the used *RETE* algorithm. The network overhead is smaller, because subscriptions may be reused in the system.

Hermes is a scalable P/S-middleware similar to *Siena*, see Section 3.3.1, by Pietzuch [123] published 2004.It uses a different approach to an event mediating overlay network by exploiting the *Pastry* DHT [137]. In contrast to *Siena's* purely content-based approach, *Hermes* uses a hybrid approach by describing events as types according to a scheme similar to *ROS'* messages. This scheme needs to be known to the middleware before communication. Filtering is possible on types of events as well as on attributes of events. The schemas of the individual events types are stored in the DHT.

Compared to *Siena*, Hermes solves the problem of the statical overlay network through the DHT. It decouples its nodes in flow and space as typical for P/S systems. Time decoupling is not implemented as events are not stored on route. The event type system enables decoupling of data structure very efficiently. Semantics coupling is created by the implicitly defined event types used to create communication channels. The context of each message only contains a time stamp. No position, uncertainty, failure or bandwidth information is provided to the application. Hermes distribution model is similar to *Siena*'s based on their overlay-networks of nodes. The DHT enables an efficient large scale routing of events based on the event types. This decreases CPU consumption, but may slightly increase bandwidth consumption in special cases. The memory consumption is still high, because based on the used hash function an storage model, the DHT present in every broker may consume much memory. Fortunately, the size of the local DHT depends on the network architecture and not on the amount of subscriptions or publications.

Hermes uses the Subscription-based filter model exploiting its event types to describe attribute filters. These filters can be combined using logical operations. O'Keeffe [115] extended Hermes' filters system with an SQL-basedone to additionally support sequence filter composition and aggregation. The goal of his extension is to enable complex event detection in a truly distributed system. Therefore, the language focuses on the use of time specification using uncertainty intervals, to cope with the lack of a reliable fine-grained global clock. Additionally, the specified language allows for decomposition of complex event statements in parameter space to limit the possible events to specific regions of the network.

The statements are parsed into a tree, which creates a hierarchical event statement that can be distributed afterwards. The system is based on Hermes' rendezvous points as complex event detectors.

Even though the author states to target WSNs with his work, the usage of an IP-based communication middleware on a DHT limits it's usage on low resource nodes. Additionally, the described operations are SQL-like and allow for simple aggregators such as `MIN`, `MAX` and `AVG`, but more complex composition operations such as digital filters or event transformation processes are hard to achieve.

Summary As visible in Table 3.5, Complex Event Detection Systems provide very good event filtering support even for sequences of events. However, the processing capabilities are limited to simple event transformations and aggregation of homogeneous events using basic operators such as *MIN, MAX* and *AVG*. All systems provide a time context to the events and some additionally, a position context, which may be used in the filter expressions. However, the systems do typically not handle QoS. Only *Snoop(IB)* and *Hermes* provide QoC using interval representation of time. While *Snoop(IB)* is centralized, the other systems are either fully distributed or only use a central master to ease consensus. The components of the systems are coupled only semantically because of the lack of a generic event type definition. The resource consumption is very heterogeneous. SENSID is very efficient, because it is already tailored towards sensor networks. HERMES uses a DHT as communication backend, which scales very well, but needs memory and CPU time to distribute the data. Snoop(IB)'s DBMS increases the resource consumption.

Criteria		Snoop(IB)	SENSID	Abstract Events	HERMES
Coupling	Time		x		
	Space	x			
	Flow	x			
	Data Type			(x)	
	Semantic	x	x	x	x
Context	Time	x	x	x	x
	Space		x	x	
	Uncertainty				(x)
	Semantics				
Quality	Service				
	Context	x			
Distribution	Type	centralized	distributed	master-based	distributed
Resources	CPU	-	++		o
	Memory	-	++		o
	Network				+
Filter	Model	SQL-based	Subscription-based	Subscription-based	Subscription-based/SQL-based
	Elementary	++	+	++	++
	Composite	++	-	++	++
	Sequence	++	o	o	++
Processing	Model	RelAlg	none	InfAlg	RelAlg
	Transformation	+	–	++	o
	Aggregation	o	–	o	+
	Fusion	–	–	o	–

Table 3.5.: Evaluation of the described CEP and CED systems against the comparison criteria, see Section 3.1.

3.5. Stream Processing Frameworks

Stream processing frameworks enable the processing of continuous data streams. They view the data stream as a continuous flow of independent events. To enable operations on this flow, they select a set of events from the stream into a window. After this transformation they can operate on the events similar to a relational database. The major use cases for these systems are monitoring applications as described by Carney [41] in 2002. These applications differ from traditional DBMS applications in the way that the active instances creating events are not human operators but sensors. This changed the focus of the system from pure storage of data in a robust and predictable way to the delivery of real-time data streams even on incomplete inputs.

Aurora/Borealis by Abadi et al. [13] published in 2005 is a Data Stream Management System (DSMS) based on Aurora [14, 15] published in 2003.

Aurora enables distributed QoS-aware stream processing based on user-specified graphs of pre-defined operators. It is very much focused on the optimization of the QoS-parameters of the systems, which are latency, data/package loss and importance. The system is bandwidth-aware by detecting overload situations and inserting artificial *DROP*-operations to the stream processing to trade data accuracy against latency.

The contained extensive query language *SQuAl* enables operations such as `filter, aggregate, union, resample` and `map` on incoming event streams. In general, three types of queries can be issued: *Continuous Query*, *Ad-Hoc Query* and *View*. The *Continuous Query* applies the defined operators directly on the events flowing through the system without any additional storage of the events. The *Ad-Hoc Queries* use persistence blocks storing events for longer time to enable the delivery of arbitrary data. A *View* enables the declaration of an event stream, which is processed by the system, even if no application is currently requesting the resulting output. This enables event stream computations which need setup time to deliver data as soon as an application needs the results.

Borealis extends the *Aurora* system using concepts of *Medusa* by Cherniack et al. [46]. It focuses on the distribution of the processing operations between nodes. At the same time it aims to preserve Aurora's focus on QoS-awareness and optimization.

The systems using Aurora or Borealis are only coupled in the semantic domain, since the meaning of the individual attributes of the event tuples is not defined. Time coupling is mitigated by the persistence blocks, space coupling is mitigated by the centralized Aurora server or the neighborhood distribution of Borealis. The context of the events contains time and bandwidth, but examples also show the ability to handle positions. Quality is only considered for the defined QoS-Attributes: latency and bandwidth. Event filtering is done using the query language in an SQL-based manner. It enables arbitrary elementary predicates and the composition of these on single event streams. Filtering of multiple event streams and filtering of sequences of multiple events within one stream can only be achieved by exploiting the *aggregation* and *union* operator. The processing language is Process and follows the Box-and-Arrows concept to derive procedural data flows. The application of arbitrary map functions allow a large set of transformations of single events, whereas, the aggregation operation allows aggregation of events in a similar flexible manner. Data fusion is not directly included in the system, but may be emulated using the existing operations.

The resource efficiency of the system depends very much on the query graph defined by the user. In general, the CPU load is pretty high because of expensive processing operations such as `join`. Additionally, *Aurora/Borealis* needs to maintain two QoS graphs in the background, which creates additional overhead. The memory overhead depends on the used window size of the operations and the amount of persistence storage blocks used. The network bandwidth consumption is limited by the load shedding dropping events on user specified overload conditions.

Complex Query Language (CQL) on STREAM was developed and implemented by Arasu et al.[25, 26] in 2003. It uses the *STanford StREAm DatA Manager (STREAM)* DSMS developed by Arasu et al. [24] and Babcock et al. [32] in 2002/2003. CQL is a SQL-baseddeclarative language to create complex events out of event streams. To this end, CQL formally disambiguates between streams and time-varying relations, which both are described by schemes defining the contained data formats. The contained operators are classified into *stream-relation*, *relation-relation* and *relation-stream* ones. *Relation-relation* operations are time-varying versions of the classical SQL-operators such as `WHERE, SELECT` and `FROM`. *Stream-relation* operations consist of a sliding windows extracting time-varying relations out of a tuple stream. CQL considers three types of these operators: *time-based, tuple-based* and *partitioning*, which is similar to SQL's `GROUP-BY` statement. The *relation-stream* operators enable the insertion or deletion of tuples into the output stream.

The implementation of CQL on *STREAM* is based on *tagged tuples*, which are tuples annotated by a time stamp and a deletion or insertion mark. This approach enables the system to keep track of the whole history, while still providing a notion of time and time-varying relations and streams. The system is centralized and enforces its input to either be in First-In First-Out (FIFO) order or to provide an explicit *heartbeat* to tell the system up to which time stamp tuples may be processed.

Systems using CQL are uncoupled in time, since any arbitrary historical data can be retrieved, but they need to know the CQL server to issue their queries. There is no information available if queries block the issuing node or not, so flow coupling is unknown. Data structure is uncoupled using the schema description of streams and relations. The context of the tuples is limited to time and no quality awareness of service or context is available. The provided filter expressions are SQL-basedand provide elementary filters against constant values. The native composition is limited to logical connectors. More complex filters can be emulated using the partition windowing and aggregation, which is also necessary for filtering sequences of events. The processing language constructs Windowed Relational Algebra (WRelAlg), which enables transformations using attribute assignment and natively supports aggregation. Data fusion is not considered in the language.

The integration in the *STREAM* system heavily relies on joins and needs to sort the stored tuples to be efficient this creates considerable overhead for the CPU. The memory consumption is quite large, since all inserted and deleted tuples stay in the database. The network overhead is not described by the authors.

Odysseus is a centralized Stream Processing System developed by Appelrath et al. [23]. The DBMS provides typical *Relational Algebra* operations such as *Selection, Projection* and *Cartesian Product*. Additionally, it enables the use of an arbitrary map function to transform events. The system is currently centralized using a single server, but authors stated it may be easily distributed. The baseline system is component-based enabling easy reuse of parts of the systems and uses Transmission Control Protocol (TCP)/IP or User Datagram Protocol (UDP)/IP as communication mechanism between components. The incoming raw data are processed and fed to processing pipes. After the processing, the results are distributed to the applications issuing the queries. The queries themselves are stated in *CQL*. Additional work was done by Kuka et al. [94] to use the system to describe higher level contextual sensors for automotive applications. For this application, the system was extended to handle events containing a time span providing some notion of time uncertainty. This approach uses a Bayesian inference system to pessimistically fuse the incoming raw events to create an occupancy grid providing a special sensor fusion operation. Even though the implementation of such fusion mechanisms is possible, they always need specially developed components to be introduced to the system and cannot be stated in native *CQL* statements.

Kuka and Nicklas [95] added the possibility to describe, process and propagate the uncertainty of individual events. To this end, they modelled the uncertainty of each event source as a Gaussian Mixture Model [111], which may either be provided by the source itself or by an estimation algorithm using maximum likelihood. The approach performed well in the described evaluation, minimizing false-positives and false-negatives in most cases. However, the necessary Monte-Carlo-based operation to combine the Gaussian Mixture Models is expensive. The map function can only multiply the existing probabilistic values of the events with static probabilities, heavily limiting the expressiveness. The authors also mentioned the need to integrate aggregation in their probabilistic uncertainty, but considered it outside the scope of their paper.

Odysseus contains a rich set of elementary filters, which can be combined very flexibly. Additionally, it allows aggregation through *CQL* operations and transformation based on the arbitrary `map` operation. However, the `map` operator and all fusion operations are outside the scope of *CQL* and need to be implemented as individual components limiting the expressiveness in these dimensions. The resource efficiency is limited by the expensive internet scale technologies such as web-services and TCP/IP communications. Additionally, all processing is executed in the centralized server. The implemented uncertainty concept is very powerful to handle uncertain events, but lacks the handling of uncertain attributes besides time. Consequently, the system considers an event context containing time, uncertainty and position. The baseline DBMS provides native decoupling of producers and consumers in time, space and data structure. A semantic decoupling is not considered in the system.

Complex Event Detection and Response (CEDR) is an event stream system by Barga et al. [35] published in 2007. It uses a very extensive temporal model containing multiple time stamps to handle uncertainty of timing information in different nodes. The three temporal domains used by the system are *system time* as used by the CEDR system itself, *occurrence time* and *valid time*. The *valid time* is used to check the validity of an event from the publisher's perspective. This enables the specification of a lifetime of each native event. The *occurrence time* specifies the last occurrence of a change in the data of an event though a processing operation. Both are specified in the time domain of the event publisher as intervals. CEDR uses this extensive time model to provide an application controllable consistency. Depending on the chosen consistency level, the behavior of the operators change. For example *Strong consistency* needs the operators to block the input stream while ordering the incoming data.

The used SQL-based language enables a rich set of filter expressions supporting complex composition of arbitrary elementary filters. In contrast to typical window-based filter languages CEDR has no window operator, but embeds the window specification in the dedicated operators. The `SEQUENCE` operator for example needs an explicit duration to search for specified events. The processing part of the language is limited to simple attribute assignments to events. The system uses a Windowed Relational Algebra (WRelAlg) and supports basic transformations of events and aggregation. Data fusion is not contained in the language.

CEDR queries may create an arbitrary amount of output events, if the application does not explicitly state a non-zero consumption policy. This may create large CPU, memory and network resource usage.

Semantic Streams Semantic Streams is sensor data streaming framework by Whitehouse, Zhao and Liu [168] published in 2006. It is based on *Inference Graphs* described by inference relations and queries in *Prolog Constraint Logic Programming (CLP)*. It uses a semantic description of sensor data streams through a common vocabulary. These streams can automatically be combined based on queries by applications. The system automatically uses available inference relations to match existing streams to requested streams. The result are multiple *Inference Graphs*, which use different inference relations to combine the sensor data of the input streams to the requested output stream. The system can handle contextual information regarding time, position and uncertainty. Time is handled by time stamps and latencies. The queries of the applications and the inference relations can specify constraints on these attributes to enable filtering and optimization. Additionally, position is handled by special relation to detect containment and intersection of regions. Uncertainty is expressed as a confidence value, which is suitable to detection events, but the applicability to continuous attributes of events is not described by the authors. The queries provide the ability to specify QoS constraints, which are used to select appropriate *Inference Graphs* or search the best graphs regarding a certain QoS metric such as latency. The constraints attached to queries and inference relations enable a Pattern-based filtering system, which provides good elementary filter support, but only supports limited composition operations. The contained information of the streams is unspecified and the inference will fail automatically if a stream does not provide the necessary information. The system can also handle run time information and adapt its behavior regarding different QoS parameters like event frequency. The system is implemented using a centralized *Prolog* interpreter, which is fed by multiple low-resource embedded sensor nodes. The authors did not describe how inference relations are implemented, which disable an estimation of performance of the CPU and memory usage. As the events need to be forwarded to the central *Prolog* instance the network resource consumption might be high depending on use-case. The authors stated the time to generate an *Inference Graph* to be minutes, which implicates a large overhead in CPU resources to create a communication stream between sensor and applications. The system especially enables mutual dependency of inference relations, which may create infinite inference loops further decreasing performance. The performance is mitigated by the inherent capability of the system to share the inference relations between different requests served at the same time. This enhances CPU performance and decreases memory consumption as no duplicate events need to be kept and processed in the system. The system considers no QoC.

Summary As visible in Table 3.6, Stream Processing Systems provide extensive processing and filtering capabilities. *Semantic Streams* is more limited regarding the filter capabilities because of its used Prolog CLP(R) expressions. However, the necessary abstraction used to implement these consume a lot of resources. Besides Aurora/Borealis and *Semantic Streams*, all systems enable processing using join operation, which is executed on windows of streams of events. This operation is very expensive. *Semantic Streams* uses inference relations, whose expressiveness depends on the amount of implemented relations. *Borealis* is the only fully distributed system in this context. The other systems are implemented in a centralized way to provide better consistency guarantees regarding the detection of composed events. Depending on the envisioned application, these systems provide different context information attached to the tuples. Time is present in all systems, but position, uncertainty and bandwidth information is present only in some of the systems. QoC is only considered by *Odysseus*, whereas, QoS is considered only by *Aurora/Borealis* and *Semantic Streams*. Semantic coupling is present in all systems because of the undefined structure of the tuples besides *Semantic Streams* which uses a vocabulary of streams and inference relations. The tuples are defined by the application developer and incorporated as schema. Since different application designers may generate different schema, a semantic translation is not easily possible. Flow and Space coupling is present in all the centralized systems, again *Semantic Streams* is an exception as it decouples sensors and applications regarding all but data types.

Criteria		Aurora/Borealis	CQL/STREAM	Odysseus	CEDR	Semantic Streams
Coupling	Time					
	Space		x	x	x	
	Flow		()	x	(x)	
	Data Type					x
	Semantic	x	x	x	x	
Context	Time	x	x	x	x	x
	Space	(x)		x		x
	Uncertainty			x		(x)
	Semantics					x
Quality	Service	x				x
	Context				x	
Distribution	Type	distributed	centralized	centralized	centralized	centralized
Resources	CPU	-	−	−	-	-
	Memory	o	−	−	-	−
	Network	+		-	o	+
Filter	Model	SQL-based	SQL-based	SQL-based	SQL-based	Pattern-based
	Elementary	++	o	++	++	o
	Composite	+	o	++	++	-
	Sequence	o	+	+	++	-
Processing	Model	Process	WRelAlg	RelAlg	WRelAlg	InfAlg
	Transformation	++	o	+	o	o
	Aggregation	++	+	++	o	o
	Fusion	o	−	o	−	o

Table 3.6.: Evaluation of the described Stream Processing Systems against the comparison criteria, see Section 3.1.

3.6. Context Frameworks

As discussed in Section 2.1.3, flexible CPS need awareness of the data's context in addition to the data. Therefore, a number of context-aware development frameworks for mobile dynamic networks were created, which define operations based on sensory data and the current context of the system.

Solar is a *Context Fusion Network (CFN)* by Chen , Li and Kotz [45] published in 2004. It requires users to write explicit context fusion operators, which merge sensory data based on the current context of the system. These operators are combined to a pipe connecting a sensor (a data source) to an application (a data sink), processing the sensory data in between. The individual operators act as event sink and source. They consume sensor data on input ports and produce new sensor data on outgoing ports. An input port can be connected to an output port of another operator using a *channel*. *Channels* can either be push-based, the data is passed as events towards the application, or pull-based, the data is explicitly requested from the application. The disseminated data are *records* containing tuples of tag-value pairs. The operators are executed on the peers of a structured overlay network implemented using the Pastry DHT [137]. Applications can discover peers as well as sensors using services provided by the overlay network. Operators are requested by applications and passed to a *Fusion* component on a peer creating an operator graph. The operator graph is checked for already existing sub-graphs in the network and uses them. This reduces the amount of computation needed because results of operators are shared between peers. Additionally, Chen and Kotz developed an alternative multicast-based transport called PACK [44]. It provides the ability to describe data reduction policies in case of overflowing routing queues within the nodes. These policies allow the node to drop specified events or aggregate them to save network bandwidth.

Solar uses XML descriptions to enable applications to select operators, queries and policies. However, the implementation of the operators needs to be done explicitly. Queries are limited to the selection of specific tags contained in a record, but using the policy system, a content-based selection can be used, if an appropriate policy was implemented. The context fusion operators can contain arbitrary code, making them very powerful, but they need to be explicitly implemented and contain no means of configuration. This enables reuse of the operator, but even a slight change in behavior requires a newly implemented operator. The context of the records only consists of bandwidth. The system contains no inherent capabilities to handle time, position or uncertainty. The resulting systems are only partially coupled in semantic, since the semantics of the records are defined by the service providing them. Depending on the extend of the machine-readable service description, a semantic selection of services may be possible. The resource consumption of the system is limited for network bandwidth due to the aggregation and dropping policy. These policies may also be used to distribute load and enable load shedding providing limited QoS. The CPU consumption is medium through the distribution of the operators and the policies, but the used technologies such as XML and HTTP increase CPU resource usage. The memory consumption is medium to heavy depending on the amount of records stored in the DHT and the amount of peers existing in the system.

Sentient Objects are an abstract programming model developed by Fitzpatrick et al. [64, 37]. It focuses on the notion of a sentient object as an autonomous, proactive and event-triggered entity within a complex networked system. A *Sentient System* is composed of *Sensors* transforming real world phenomena to events, *Actuators* transforming events to real world phenomena and *Sentient Objects* computing and processing events. The main goal of this abstraction is the reuse of defined components to ease the development of complex applications.

The output of a *Sentient Object* depends on its current context, where context is defined as the combination of all relevant sensory information. The generation of the necessary context is divided into two steps. First a Bayesian network is used to fuse incoming sensor event data in a robust manner to provide low-level contexts. Afterwards, low-level contexts are combined using a rule-based inference system. The possible contexts are pre-defined by the developer of the *Sentient Object* and are heavily mission specific. To increase the performance of the system, only a single context in each *Sentient Object* is active at a time. Consequently, only the relevant sensory data for this context are fed to the Bayesian network increasing the efficiency.

The used Scalable Timed Events and Mobility (STEAM) middleware provides a P/S-based filtering of events based on topic, content and position. The Bayesian networks provide inherent data fusion capabilities even for uncertain data. However, the relevant networks need to be constructed through a GUI by the developer of the *Sentient Object*. The rules of the context fusion are also constructed using a GUI by the developer.

The resulting *Sentient Systems* are decoupled in space, flow and data structure providing each *Sentient Object* with a context containing at least time, position and uncertainty. The system is fully distributed. A demo application of an autonomous sentient vehicle using *Sentient Objects* by Sivaharan et al. [149] is developed using iPAQ embedded PCs as basis. Consequently, the efficiency optimisations of the Bayesian networks and the context inference limit the CPU resource consumption. The application uses 11 MBit WLan and a 1Mbit CAN-Bus showing the ability to work with limited bandwidth networks. The memory consumption is not described by the authors. QoS can be described as declarative properties attached to the event subscription. Verissimo and Casimiro [165] described how to enforce timeliness as a QoS attribute for *Sentient Systems*. QoC is considered implicitly using the Bayesian network to handle noisy sensor data.

ACTrESS is an extension to the *Java Message Service* API by Freudenreich et al. [67]. It is a context framework aiming to describe sensor data as contextual messages. The context consists of timing, positional and value information accompanied by the respective unit and scaling information. To this end, it uses the *Java Message Service* API on top of an overlay networked P/S system as communication backbone. The authors identified the problem of the context of individual messages being incompatible and solved this using a hierarchy of contexts with a dynamic context translation. However, they also stated that there is no general root context, which is always the top of the hierarchy. Therefore, they infer the hierarchy dynamically and translate all received contexts to and from this inferred root context. The transformation is composed of basic arithmetic operations on the contained attributes. The defined context directly induces the necessary operations such as e.g. rescaling, unit conversions or change of reference system. The system uses a transformation engine in each broker implemented on top of ActiveMQ [150] to dynamically translate incoming messages. Additionally, each broker contains a context description cache to decrease network bandwidth needed to gather the necessary information for each transformation.

The nodes of the system are only coupled by the semantics of the contextual message, since all other couplings are abstracted by the used P/S backbone and the *Java Message* API. The contextual information is rich containing time, position and unit and scaling information. The system is fully distributed using the P/S overlay network. The resource consumption is medium concerning CPU and memory because of the used *Java* system. The used network bandwidth is rather small because the *Java* Message API uses a binary serialization mechanism that creates serialized representations with little overhead. Since Javas byte code is the same on all platforms, no further conversion is necessary. Because of the used P/S system, a subscription-based filter model is used, which only allows filtering using topics. Therefore, only limited elementary filters are possible. The transformation model is process-based using dynamically generated context transformations. These allow the transformation of one event from one context to another. Consequently, no aggregation or fusion is possible using this system.

Semantic Markov Logic Networks (SMLN) described by Mohammed et al. [113] enable logical inference on events in CPS. The goal is to do CED in a scenario specific context. The described approach uses an ontological context description, where the context is separated in a generic part called *Base Ontology* and a specific part called *Extended Ontology*. The ontologies are used to describe an SOA composed of agents, which collaborate and communicate using the ontologies. Agents using different *Extended Ontologies* can only communicate through the *Base Ontology*, whereas, agents using the same *Extended Ontology* can also communicate and collaborate through them. The used SOA uses no service lookup mechanism, but explicitly needs specification of the agents' *deployment* to direct the service requests to the receiving agent. The contextual representation is dependent on scenario, but contains at least a time stamp, a unit and a *Unified Resource Identifier*. The inference is executed through a *Markov Logic Network* [132] using fuzzy membership variables instead of probabilities to enable the handling and expression of uncertainties of the sensor data. The *Markov Logic Network* needs training to be able to classify incoming events correctly and to output detection events accordingly. The authors tested the system with an automotive and a smart home test data set. In these data sets they achieved between 60% and 70% of correct classification. The system was never deployed or tested in real CPS.

The proposed system provides strong couplings in time and flow because of the used SOA. It enables distributed computing using the agent-based communication and collaboration without central coordination. However, the necessary deployment attribute used to address the individual agent's services establishes space coupling. Even though the system allows the specification and processing of uncertainty in the sensor data through the fuzzy inference process, it provides no means to specify and handle QoS and QoC. The CPU, memory and network utilization is unknown, as the system is not tested in a real environment and the authors did not provide any results on these metrics. Filtering is executed through the *Markov Logical Network* and the specified inference operations. It allows for elementary filters, but no notion of sequential filters exists. Composite filters can be expressed through logical combination of elementary filters in the network. The processing is done solely through the inference network and only allows event-based reasoning without any data processing. Consequently, only transformations are generally expressible. Aggregation can only be expressed towards uncertainty increasing the reliability of the results. Data fusion in the sensor processing sense cannot be expressed.

Summary As visible in Table 3.7, *Context Frameworks* provide very heterogeneous attributes. They decouple the components very well regarding space, flow and data structure. *Solar* additionally provides a decoupling in time and partially in semantics using service description mechanisms. *ACTrESS* mitigates the semantic coupling for the position and time context of the exchanged data using implicitly generated context trees. The context definition is heterogeneous as well, Solar focuses on bandwidth as context, while *Sentient Objects* and *ACTrESS* provide time and position as context. *Sentient Objects* handle uncertainty as a context attribute using an incorporated *Bayesian Fusion Network*. The resource consumption is larger than *WSN Middlewares* but still lower compared to the *Stream Processing Frameworks*. The filter and processing expressiveness is very heterogeneous as well, depending on the envisioned usage. *Solar* provides good aggregation to handle network overload, *Sentient Objects* provide implicit data fusion using the Bayesian Fusion Network and *ACTtrESS* provides very good transformation between event contexts. A very strong point is the handling of QoS and QoC in *Solar* and *Sentient Objects*, which most other systems cannot provide.

Criteria		Solar	Sentient Objects	ACTrESS	SMLN
Coupling	Time		x	x	x
	Space				x
	Flow				x
	Data Type				
	Semantic	(x)	x	(x)	
Context	Time		x	x	x
	Space		x	x	x
	Uncertainty		x		x
	Semantics				x
Quality	Service	(x)	x		
	Context		x		
Distribution	Type	distributed	distributed	distributed	distributed
Resources	CPU	o	o	-	
	Memory	-	()	−	
	Network	o	o	+	
Filter	Model	Subscription-based	Subscription-based	Subscription-based	Pattern-based
	Elementary	+	+	-	+
	Composite	+	+	−	+
	Sequence	o	−	−	−
Processing	Model	Process	InfAlg/Rule	Process	InfAlg
	Transformation	o	-	++	o
	Aggregation	+	-	−	-
	Fusion	o	+	−	−

Table 3.7.: Evaluation of the described Context Frameworks against the comparison criteria, see Section 3.1.

3.7. Conclusion

The review of existing solutions in this chapter showed that existing systems can only provide a partial decoupling of components of CPS. The SDF provide very extensive mechanisms for the description of the sensory data and their context. Additionally, they provide mechanisms to detect, select and integrate sensors dynamically into systems. However, these frameworks lack extensive capabilities to filter and process or distribute events created by the sensors. Classical P/S middlewares enable a scalable and efficient distribution of events. They typically provide event filtering capabilities, but only limited processing. Additionally, only some are usable on embedded devices used in WSN and CPS. Special middlewares for WSN enable very efficient communication suitable for embedded devices. However they are designed for good processing capabilities, mainly focused on aggregation, or for very low resource consumption. A special type of WSN middlewares uses ontologies to dynamically translate events between semantic domains. They provide semantic decoupling between the components forming the system, but typically need additional resources. CED systems enable a distributed filtering and processing of events using expressions described by the consuming applications. They provide good decoupling in space, flow and data structure. Stream processing frameworks contain the richest set of event filtering and processing capabilities, but are often implemented in a centralized way to provide strong consistency guarantees. The used operations performing the processing and filtering are expensive and require additional resources. An exception is *Semantic Streams*, which limits resource consumption to the stream generation phase and provides more efficient *Inference Graph*-based processing and filtering after the stream is established. The last category of systems reviewed are *Context Frameworks*, which focus on the description and processing of the context of the disseminated events. However, they either enable a very extensive implicit context processing or focus on large scalability.

The reviewed systems already provide a good decoupling of components in space, flow and data structure using P/S communication. Time decoupling may be implemented through additional storage facilities. Semantic decoupling is very difficult to achieve. Therefore many systems do not handle it at all. Some systems such as *LooCI* and *Semantic Streams* provide solutions using local vocabularies of sensors, which are used in automatic inference of processing. The context descriptions of events in time is generally available. Positional context information is dependent on the envisioned application. Uncertainty of events is only handled by a small part of the systems mostly as an extension. QoS is present in some systems and focuses on package drop and latency. QoC is very rarely tackled in the systems. Filtering and Processing is extremely heterogeneous and depends strongly on the application, the system was tailored to. In general, good support for filter expressions decreases the necessary network bandwidth used. In contrast, powerful processing operations need many resources to be executed and create possibly unbounded output overloading the network. *LooCI*, *Semantic Streams* and *ACTrESS* are interesting systems, because they together fulfill most of the criteria. They still lack time decoupling , support for full uncertaintz specification and processing and full QoC support. Some of them also need too many resources to be used in small embedded systems. Consequently, the goal of this thesis is the development of a system fulfilling these missing properties, while still maintaining the properties of these three systems.

4. Abstract Sensor Event Information Architecture (ASEIA)

Following the analyses of the state of the art in Chapter 3, this chapter presents the ASEIA to tackle the remaining challenges. The next section summarizes the challenges ASEIA needs to solve, based on existing solutions from Chapter 3. The chapter continues with the description of the the general architecture in Section 4.2. Afterwards, the context attributes and their descriptions are explained in Section 4.3. The main part describes the ASE model used in ASEIA in Section 4.4 and the different types of transformations in Section 4.5 with different examples in the context of the scenarios of Section 1.2. The final part of the chapter provides mechanisms to select and execute these transformations within the CPS on demand. The chapter closes with a summary in Section 4.7

4.1. Goals

The ultimate goal of dynamic composition of CPS on run time can only be achieved by minimizing the coupling between the components of the CPS. The secondary goal of dynamic adaptation to the environment is achieved by the same means of decoupled components changing their composition at run time. To enable a decoupling of components of CPS a solution for the domain specific processing of events and the specification of their context in uncertain environments needs to be found.

As described in Section 2.4, ways to decouple the components according to Time, Flow, Space, Data Format, Semantic and Context need to be found.

Context decoupling is realized by a context description which is able to handle uncertainties and provides a QoC metric that enables CPS applications to evaluate the benefit of the contained information. Additionally, the events themselves need to be considered as uncertain, because the existence of an event already conveys information. Furthermore, the system needs to enable processing of events automatically. This allows an automatic adaptation of components' interfaces to each other to automatically fulfill requirements of CPS applications. The automatic processing needs to process context information attached to the events to enable adaptation of context. The definition of context, QoC and the automatic processing of this data is not well tackled by the systems described in Chapter 3.

Flow and Space decoupling may be achieved by choosing an appropriate communication paradigm. P/S enables the decoupling of components in space and flow, which is a good basis for the system, as described in Section 3.3.1.

The semantic decoupling can be achieved by using semantic enhancement of the event description either through ontologies or through LooCI's, see Section 3.3.3, approach of a unified event vocabulary.

Data Format decoupling may be achieved through a machine readable event meta-description as used by Hermes, which allows an automatic analysis of contained values of physical phenomena. The components can use these information to automatically select the information necessary for their implemented behavior.

As the system needs to be usable on a wide variety of system ranging from low-power micro controller used for WSN to full-scale PC-architecture used to monitor and control industrial processes, the used abstractions a mechanism need to be designed with low resources in mind. Especially the encoding and processing of the meta-information on context, data format and semantics need to be light-weight enough to be handled by embedded systems. This challenge is not sufficiently solved by any of the systems described in Chapter 3.

The resulting system needs to be able to execute complex sensor data transformation, aggregation and fusion operations to provide the applications with the necessary data. Additionally, the available and generated sensor data needs to be filtered early in the system to prevent overloading the network, CPU and memory resources of the nodes. The SQL-based systems already provide flexible filter operations usable to this end. *Aurora/Borealis* provide these operations with the addition of good transformation and aggregation capabilities. However, sensor data fusion operations are only present in very few systems like *MOSAIC*, *SensorML* and *IEEE1451*, because most systems aim at a higher level of abstraction.

The shortcomings of the systems discussed in Chapter 3 shall be solved by the systems described in this chapter. The systems shall fulfill the following goals:

Data Format Decoupling The system shall automatically translate between different data formats to enable independent development and specification of the components of CPS.

Context Description The system shall enable a definition of the necessary context to enable automatic processing of the whole sensor information. The context shall be described regarding the semantics, data format and quality.

Data Fusion Operations The quality (QoS and QoC) information of sensor data is a necessary parameter for any CPS application. Specific existing data fusion operations shall be available to the applications as processing mechanism adapting the sensor data and the context.

Ressource Efficiency The system shall be tailored towards low-resource nodes to enable maximum deployability towards generic CPS.

Distribution The system shall natively support distribution of acquisition and processing of sensor information.

4.2. General Architecture

ASEIA considers a fully distributed system using P/S as communication paradigm. Consequently, each node in the system fulfills on of the following roles:

Publishers generate events based on their input data or their observations of the environment and pass these to brokers or subscribers.

Subscribers receive the events generated by either publishers or brokers and pass them using an API to the CPS application.

Brokers forward and process the incoming events to fit requirements of subscribers to the provisions of publishers.

The network of the system is considered to change because of failures of individual nodes or the movement of nodes. However, the design and implementation of the actual P/S communication is not handled by ASEIA. ASEIA only uses an existing P/S system to provide enhanced functionality and abstractions. The full distribution of nodes without any centralized component induces additional synchronization functionality. These functionalities are the matching of publisher and subscribers to create communication channels, clock synchronization between the nodes as well as synchronization of references of different physical phenomena such as origins and orientations of coordinate systems or zero points of temperature scales. This contextual information is handled by ASEIA through machine readable descriptions, transformations and automatic inference mechanism.

ASEIA is designed to be an extension to an existing P/S-System. The Figure 4.1 shows the different components of ASEIA and their integration in the typical event flow of a P/S-System. The system consists of the following components:

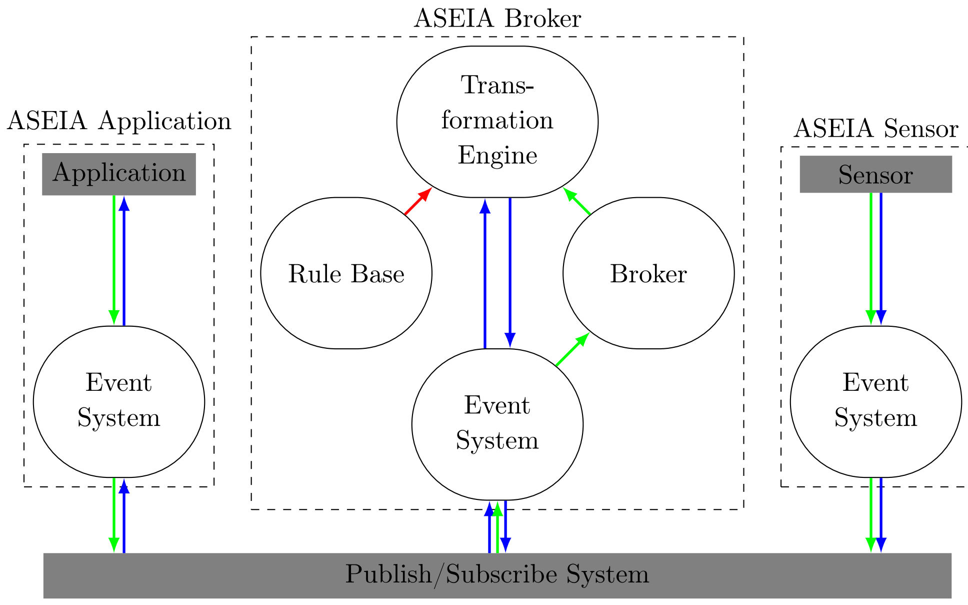


Figure 4.1.: An overview diagram of the general ASEIA Architecture. Blue lines represent *Sensor Events*. Green lines represent *Channel Events* and red lines indicate *Transformations*.

Publisher Adapters extend the Publishers of the basic P/S according to the system model, which adds format, semantic and context descriptions of the contained information in a machine readable way.

Subscriber Adapters extend Subscribers of the basic P/S system. They extend the API used by CPS systems with the ability to specify requirements on the information regarding data, format, semantics and context.

ASEs extend the basic Events of the P/S system by a machine readable description of the context of the contained information to enable automatic processing.

Transformations are operations on ASE that modify the semantics, format, context and/or content. They are described to be automatically executed if necessary.

Event Types denote the semantic and format meta-information used by the system to evaluate compatibility of publishers and subscribers. It is also used to evaluate the usability of transformations.

Knowledge Base is the storage of existing transformation rules available to transform ASEs.

Transformation Engine is responsible for the inference of transformations necessary to establish a channel between Publishers and Subscribers. Additionally, it executes the transformations on ASE reception.

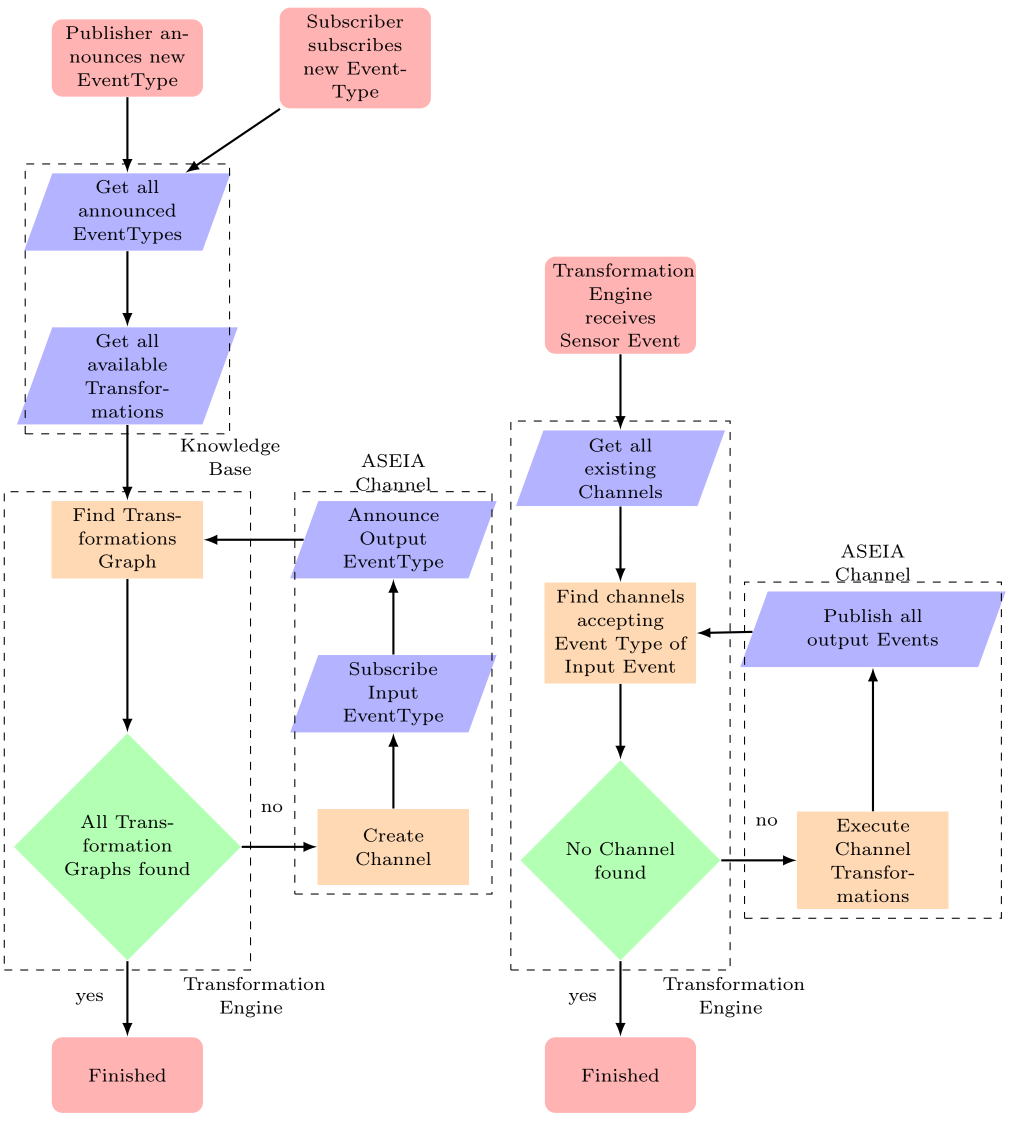
ASEIA Channels are combinations of existing transformations that accept incoming ASEs of special *Event Types*. The incoming events are forwarded to the internal transformations to output ASEs fulfilling the requirements of the associated subscribers.

ASEIA-Brokers extend the P/S brokers with the capability to execute *Transformations*, while ASEs are routed from Publisher to Subscriber. The necessary transformations for a Publisher-Subscriber-Combination are stored in ASEIA-Channels.

The general architecture consists of an abstract *ASE System* providing a type hierarchy of ASE together with (de)-serialization and processing facilities. ASEs may be transformed and aggregated by using *Transformations* abstracting generic sensor and information processing mechanisms.

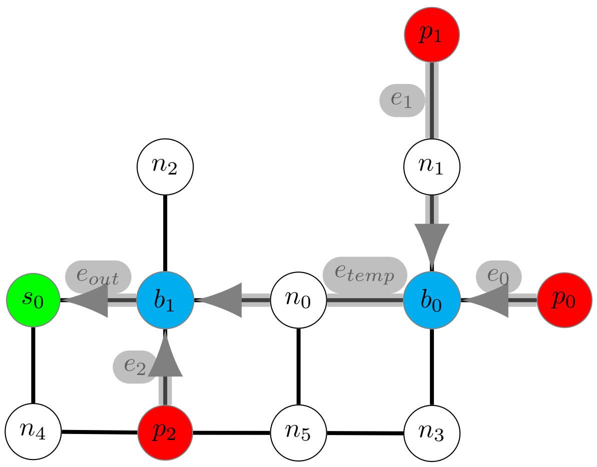
The system uses the existing P/S-Mechanism to communicate *Channel Events* and ASEs. *Channel Events* containing meta-information on an ASE are automatically published by ASEIA-Publishers and ASEIA-Subscribers on creation. In each broker node of the underlying P/S-System an *ASEIA-Broker* is present. It handles the *Channel Events* of *ASEIA-Publishers* and *-Subscribers*. It forwards this information to the *Transformation Engine*, which searches for combinations of relevant *Transformations* in the *Knowledge Base* that produce the required ASE. This search is executed whenever a new *Event Type* is used in the system. Afterwards, the resulting *Transformations* act as subscribers and publishers themselves and automatically transform incoming ASEs. The process of handling announcements and subscriptions is visualized in Figure 4.2a. On reception of an ASE compatible to the input *Event Types* of a created channel, the event is fed to the *Transformations* of the channel to produce output ASEs, which are published. This process is visualized in Figure 4.2b.

The *ASEIA-Brokers* can be distributed in the network. Multiple brokers on route between publisher and subscriber may execute separate parts of the possibly combined *Transformation*. In case no *ASEIA-Broker* exists on route, the *ASEIA-Subscriber* itself acts as a

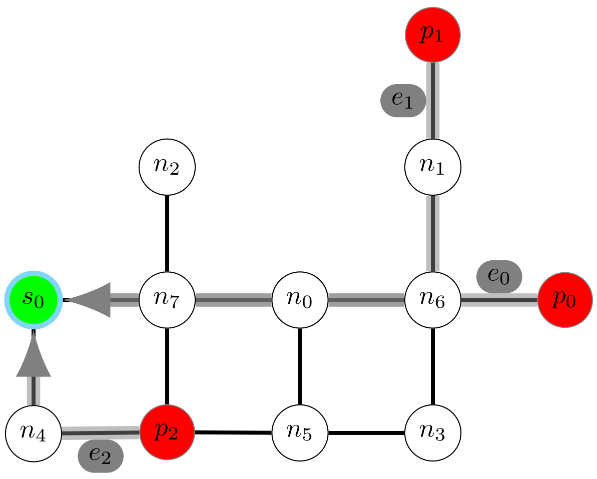

Publisher announces new EventType
Subscriber subscribes new EventType
Get all announced EventTypes
Get all available Transformations
Knowledge Base
Find Transformations Graph
ASEIA Channel
Announce Output EventType
Subscribe Input EventType
All Transformation Graphs found
no
Create Channel
yes
Transformation Engine
Finished
Transformation Engine receives Sensor Event
Get all existing Channels
Find channels accepting Event Type of Input Event
ASEIA Channel
Publish all output Events
No Channel found
no
Execute Channel Transformations
yes
Transformation Engine
Finished

(a) ASEIA Channel Creation. (b) ASEIA Channel Execution.

Figure 4.2.: Flow charts showing the behavior of the ASEIA Broker on ASEIA Channel creation and Sensor Event reception.



(a) Illustration of an example network enabling the transformation of input events e_0, e_1, e_2 to output event e_{out} by two intermediate brokers.



(b) Illustration of an example network enabling the transformation of input events e_0, e_1, e_2 to output event e_{out} without intermediate brokers.

fall-back broker. The following Figures 4.3a and 4.3b show an exemplary event flow from publishers p_0, p_1 and p_2 to subscriber s_0. The first figure shows the usage of distributed brokers b_0 and b_1 transforming the input events e_0, e_1 and e_2 to intermediate event e_{temp} and finally to the output event e_{out}. The second figure contains no dedicated brokers and all events are delivered to the subscriber s_0 which acts as fall-back broker and executes the *Transformations* internally before delivering the result to the application.

The following section discusses the context attributes which this thesis assumes necessary for generic CPS.

4.3. Sensor Information and Context

In order to use raw sensor data in a CPS application, a transformation from data to information is necessary, as described in Section 2.2.2. This process enhances the data produced by the sensor. The result is sensor data accompanied with contextual data and semantic data, which represents the inherent sensor information. The context of the sensor data enables subscribing applications to assess the usability of the data regarding its own needs as well as reasoning on abstract sensor information, as described by Zug [172].

It is difficult to estimate the necessary context of sensor data for an arbitrary system. However, for generic CPS systems, some baseline *Attributes* can be estimated based on the typical processing operations applied to the sensor data. These operations are typically executed in the *Cyber Layer*, see Section 2.2.3 or in the *Cognition Layer*, see Section 2.2.4 of a CPS. The contained operations are tempo-spatial computations and comparisons on the data as well as estimations and improvements of data quality. The concrete operations executed are discussed in Section 4.5.

The resulting contextual information of this analysis can be separated into *Event Context* and *Attribute Context*. The *Event Context* contains attributes that are necessary to identify, reason and process the sensor data in the CPS. It at least contains information on time, position and origin of the sensor information. The *Attribute Context* enhances each attribute in the event with additional information intrinsic to the process in which the information is used. In classically developed systems this attribute context is handled implicitly by the engineer or is defined prior to the actual implementation. It contains information on units, scaling, references and uncertainty. The individual context attributes are described in the following sections.

4.3.1. Attribute Context

The *Attribute Context* consists of four parameters: *Scale*, *Unit*, *Reference* and *Uncertainty*.

The unit of an attribute enables the representation of a physical phenomenon's context. Typically, the Système international d'unités (SI)-System assigns each physical phenomenon a certain *Unit*. The *Unit* parameter enables applications to further assess the content of attributes contained in events. Mathematical operations on attributes need to take the *Unit* into account and modify it accordingly. This enables additional checks to prevent misuse or erroneous operations. In contrast to other *Attribute Context Parameters* the *Unit* of an *Attribute* cannot be transformed explicitly.

The *Scale* is part of the *Unit* in the SI-System. However, ASEIA handles the scale as a separate context parameter. It assigns each attribute value a strictly monotonous function that modifies the value. It enables the designer of a CPS component to optimize the used data types in processing and communication to the scenario. An example is the usage of the *milli* scale, which multiplies each value with 1/1000. It enables a phenomenon, which contains values v in the range $-1 < v < 1$, to be meaningful transported as integers.

The *Reference* attribute contains the reference of measurement of the phenomenon. This context parameter enables different measurements with different mechanisms to be used together in computations. Examples are temperature measurements in Kelvin and Degree Celsius. The values are compatible regarding mathematical operations if they are transformed to the same reference point. In classical systems, reference points are typically fixed based on the pre-defined SI-Unit definitions. ROS, see Section 3.3, for example defines each value communicated to be in basic SI-Units. However, some SI-Units define no absolute reference, e.g: Position. ROS established a dedicated system to handle the different position references, which is named *TF* [65]. *ACTrESS*, described in Section 3.6, dynamically creates hierarchies of different scales used for position data. This enables a translation of the sensor data produced in one scale to any other scale within the hierarchy. However, the translations between the entries in the hierarchy are executed on certain information and output certain information. In consequence, translations, which decrease the quality of the data pass silently through. This creates problems for the CPS applications using the contextual information as the usability of the contained information cannot be evaluated anymore.

Consequently, ASEIA enforces a generic attribute context parameter *Uncertainty*, which is described in the next section.

4.3.2. Uncertainty Model

An identified challenge of the dynamic composition of CPS, as discussed in Section 2.2.3, is the definition, communication and propagation of uncertainty of sensor information in the system. Currently, a major task of an engineer designing and implementing CPS is the estimation of the inherent uncertainty and its management as this uncertainty is inherent to the system and cannot be avoided as Elmenreich [61] described.

Figure 4.4 shows an example situation in the car scenario. A car equipped with a distance sensor has a certain observation area of this sensor. Within this observation area it is able to reliably detect and estimate the distance to another object. However, the exact angle of this other object is unknown as only a distance is produced. If the distance information is transformed to a position information of the other object, the angular uncertainty needs to be considered and transformed to a positional uncertainty.

A fully distributed and individually developed CPS needs to handle this process automatically by explicitly handling the existing uncertainty in all information and processes. This includes the sensor data as well as the context of the sensor information. To handle this uncertainty information, uncertainty models that enable a mathematical abstraction of the existing uncertainty are necessary. These models need to include operations to forward and modify the uncertainty through the processes executed in the CPS from source (sensor) to sink (actuator). The uncertainty of the information is dependent on the information, therefore, it is useful to have separate uncertainty models for different types of information.

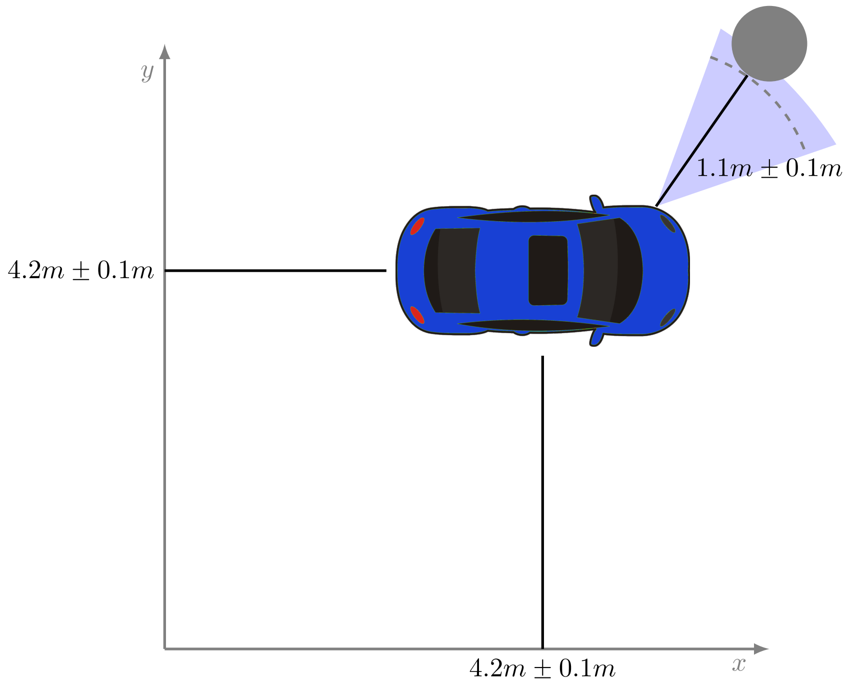


Figure 4.4.: Visualization of the time space uncertainty context of an abstract distance sensor in a vehicle scenario.

Semantic Streams, see Section 3.5 focuses on an *Uncertainty Model* for the whole event, which describes the confidence in the event's existence. *Odysseus* uses *Gaussian Mixture* models to model and process the uncertainty of the sensor data. This approach enables an accurate estimation of the uncertainty of acquired sensor data, if enough training data to fit the model can be acquired. The operations on *Gaussian Mixture Models* are *Monte-Carlo* algorithms sampling the distribution to execute mathematical operations, which is computationally expensive. *Sentient Objects* attach Bayesian beliefs to the sensor data to enable Bayesian inference on the distribution of the continuous values, which needs a known distribution of the sensor data in case of continuous data. All three approaches are valid, but handle different types of information and use different amounts and types of input data.

In this thesis two uncertainty models are considered. One model handles discrete valued information and another one handles continuous valued information. These two uncertainty models aim to provide an uncertainty representation for continuous and discrete values of sensors and context. Mathematical operations executed on the sensor data are also executed on the uncertainty and transform the contained information accordingly. The defined operations aim to induce minimal computation overhead to enable the application to low-power embedded systems. Additionally, the uncertainty models trade exactness of the uncertainty representation against easy specification and fast operations at run-time. However, ASEIA is designed to handle any type of uncertainty model as long as necessary mathematical operations are defined. The following Section shows the used discrete and continuous uncertainty models designed and later evaluated for ASEIA.

Discrete Value Uncertainty

Discrete valued data has the special property that neighboring values are independent. In a mathematical sense no distance metric exists on the data. An example is the classification of an object. The object might be classified as a cup or as a ball. However, these two results are completely independent. In general, it is not possible to deduce the probability of the object being a ball, if the object was classified as a cup. Therefore, discrete valued information uses probability-based uncertainty. The probability describes the amount of possible miss-classification depending on the value of the information.

The information uncertainty complex $v \in \mathcal{I}_X = X \times [0,1]$ can be described as a tuple of the actual information $v^v \in X$ with the associated miss-classification probability $v^u \in [0,1]$. The necessary operations for discrete values information are mainly comparison operations such as $\diamond \in \{=, \neq\}$. Consequently, the comparison operations can be defined based on the probabilities as:

$$a \diamond b : \mathcal{I}_X^2 \to \{0,1\} \times [0,1] \tag{4.1}$$

$$a \diamond b = \left(a^v \diamond b^v, 1 - (1 - a^u)(1 - b^u)\right). \tag{4.2}$$

The output of these comparisons are discrete values of booleans. These have two additional operations that enable the expression of logical connections $\circ \in \{+, \cdot\}$. The operations represent logical *or* and logical *and*. They are defined according to Equations 4.4 and 4.3. Additionally, the negation of a boolean $\neg$ is defined in Equation 4.5.

$$a \circ b : \mathcal{I}_{\{0,1\}}^2 \to \mathcal{I}_{\{0,1\}} \tag{4.3}$$

$$a \circ b = \left(a^v \circ b^v, 1 - (1 - a^u)(1 - b^u)\right) \tag{4.4}$$

$$\neg a = \left(\neg a^v, 1 - a^u\right) \tag{4.5}$$

This approach enables the propagation of the uncertainty of the different decision steps in the processes executed in the CPS. In general, the uncertainty of this information is monotonously increasing, but special transformations such as filters and aggregations may decrease it.

Continuous Value Uncertainty

In this work continuous valued uncertainty is modelled by using interval arithmetic, because interval arithmetics is a well understood tool to express uncertainty, see Dawood [55]. To enable the processing layer of ASEIA to use the uncertainty model, an algebra needs to be defined that provides the necessary operations by using operations defined on intervals. The described operations are based on the algebraic operations described in [56].

An interval is defined as a continuous subset of a domain D. In the case of computer arithmetic, this domain is generated by the possible values of the generic data types such as unsigned integer with 32 bit, signed integer with 16 bit and *IEEE754* floating point numbers [9]. An interval I may either be defined by the smallest contained value $a^- \in D$ and the largest contained value $a^+ \in D$, with $a^- \leqslant a^+$ as $a = [a^-, a^+]$ or as the center of the interval $a^v \in D$ and a distance to the bounds of the interval $a^u \in D^+$. D^+ denotes a domain related to D containing only positive values $0 \leqslant d \in D^+$. The resulting interval may be written as $a = [a^v \pm a^u]$. This representation eases the evaluation of the uncertainty and the value part of the interval. It also enables the interval to span outside of the domain D, which may be used to indicate computation errors such as negation of an unsigned value. Mathematically, both representations can be converted by using Equations 4.6 and 4.7. Consequently, this work will use the representation of Equation 4.7.

$$\left[a^-, a^+\right] = \left[\frac{a^- + a^+}{2} \pm \frac{a^+ - a^-}{2}\right] \tag{4.6}$$

$$[a^v \pm a^u] = [a^v - a^u, a^v + a^u] \tag{4.7}$$

To be able to handle intervals similar to normal arithmetic types, the operations $+, -, *, ||, ^2, \sqrt{}, <, >, =, \neq, \leqslant, \geqslant$ need to be defined. The basic arithmetic operations $\circ = +, -, *, /$ are defined through Equation 4.8.

$$a \circ b = \left[\frac{min(a \circ b) + max(a \circ b)}{2} \pm \frac{max(a \circ b) - min(a \circ b)}{2}\right] \tag{4.8}$$

The approach of Equation 4.8 can also be used to enable arbitrary mathematical functions to be applied to the interval I. If the applied function is not monotonous, the interval needs to be separated at the points of monotony change. The function can then be applied and the resulting sub-intervals need to be merged again.

The comparison operations $\diamond = <, >, \leqslant, \geqslant, =, \neq$ can be reduced to the definition of the two ordering relations $<$ in Equation 4.10 and $>$ in Equation 4.11 and the equality relation $=$, see Equation 4.12.

$$a \diamond b : \mathcal{I}_X^2 \to \mathcal{I}_{\{0,1\}} \tag{4.9}$$

$$a < b : \begin{cases} (1, 0) & \text{if } a^v + a^u < b^v - b^u \\ (0, (a \geqslant b)^u) & \text{otherwise} \end{cases} \tag{4.10}$$

$$a > b : \begin{cases} (1, 0) & \text{if } a^v - a^u > b^v + b^u \\ (0, (a \leqslant b)^u) & \text{otherwise} \end{cases} \tag{4.11}$$

$$a = b : \begin{cases} (1, (a = b)^u) & \text{if} \min(a^v + a^u, b^v + b^u) > \max(a^v - a^u, b^v - b^u) \\ (0, 0) & \text{otherwise} \end{cases} \tag{4.12}$$

$$(a = b)^u = \frac{(\min(a^v + a^u, b^v + b^u) - \max(a^v - a^u, b^v - b^u))^2}{4a^u b^u} \tag{4.13}$$

$$(a \geqslant b)^u = 1/2(a = b)^u + \frac{\max((a^v + a^u) - (b^v + b^u), 0)}{a^u} \tag{4.14}$$

$$(a \leqslant b)^u = 1/2(a = b)^u + \frac{\max((b^v - b^u) - (a^v - a^u), 0)}{a^u} \tag{4.15}$$

The order relations $<$ and $>$ provide an exact output if the intervals in question are not intersecting. Otherwise, they provide a measure of the ratio between intersecting and accurate parts of the intervals. The defined equality comparison $=$ represents an intersection test relation between a and b. A minimum uncertainty is only established if the intervals are exactly equal. Otherwise, the relation defines an integrated measure of intersection area. The other comparison operations can then be expressed in terms of $<, >$ and $=$.

$$a \leqslant b : \neg(a > b) \tag{4.16}$$
$$a \geqslant b : \neg(a < b) \tag{4.17}$$
$$a \neq b : \neg(a = b) \tag{4.18}$$

4.3.3. Time Model

Time is a very important attribute of any information used in CPS. As described by Kopetz [89], the information and the time of creation, respectively acquisition, form a time-value entity, which cannot be separated during the whole processing. Since time is considered to advance monotonously, information can only become older, unless new information is acquired. This process of aging is very relevant to any CPS since the age of the information is directly coupled to the uncertainty of the information. In general, older information induces additional uncertainty to the information. The actual coupling between age and uncertainty is very system-specific and is typically not known. However, special operations such as *Kalman Filters*, as described in Section 4.5.8, may use explicit information on the time uncertainty coupling of a specific process to minimize the uncertainty of the information.

As stated by Einstein [87] time is a local attribute, which cannot be observed globally. Consequently, each component in the system has its own local time. Even if clock synchronization is applied, a residual uncertainty still persists. Clock synchronization is generally used to synchronize the clocks of the different components to a common value until a desired uncertainty goal. For an arbitrary large system the estimation of this common value becomes more difficult and might induce additional uncertainties. Existing clock synchronization algorithms are typically bound to certain network topologies and/or enable a static trade-off between uncertainty and overhead. For example *Continous Clock Synchronization (CCS)* [112] generally considers single-hop neighborhoods to ensure minimum uncertainty. Caesium Spray [166] on the other hand uses a second network level consisting of GPS-receivers to limit the number of hops. Other approaches such as the one proposed by Römer [135] modify the time stamps on transmission of the information from sender to receiver. This approach scales well, but induces statically unknown uncertainties.

This thesis uses a time model that is based on local time domains in each node, which is similar to *GSN*, see Section 3.3.2. In addition to to the approach of *GSN*, translations between the individual node's time domains are estimated by observing the time of neighboring nodes. This enables the node to assess the difference between the time domains up to the inherent uncertainty of the system. This difference of time domains may be used to transform information with a local time stamp from one domain to another, which relates to *ACTrESS* approach, but applies it to the time domain. However, the estimation can never be fully accurate, therefore, the transformation increases the uncertainty of the time information of the transformed *Events*. Information flowing through the system is transformed from one time domain to the other depending on its current topological position in the system. On each transformation the uncertainty increases, resulting in smaller uncertainties for topologically closer time domains and larger uncertainties for topological distant nodes. The used *Transformation* to implement the used *Time Transformations* is described in Section 4.5.9. A consequence of the used *Time Transformation* system is the definition of the time context of an attribute itself as a time stamp with an attached uncertainty. This uncertainty naturally defines a time-span instead of a certain time stamp, as done by e.g. *Sensid*. Therefore, time spans can be expressed using an uncertainty in the events time context. Finally, events are considered to exist not at a certain point in time, but have a minimum start time and a maximum end time.

A large benefit of this approach is the locality of the time information, which eases the processing of sensor data. Additionally, time stamps are inherently considered as time spans, which enables a native uncertainty propagation during processing. Comparison of time stamps can yield a third result *maybe* describing the inability of the system to exactly establish an order. This concept is similar to the approach of Römer [135]. The resulting time attribute uses one dimensional values with continuous uncertainty, see Section 4.3.2, which supports the classical mathematical operations for scalars $+, -, *, /$ and the typical comparison operations $<, \leqslant, =, \neq, >, \geqslant$. The *maybe* ordering is realized by the definition of the comparison operations, which contain different semantics compared to certain comparison. The unit of time is generally *Seconds* with an arbitrary linear scaling based on the precision of the local clock of the node. The reference of time can be an arbitrary point in time such as the Unix epoch 00:00:00 1.1.1970. Typically, each node has its own reference point, which gets transformed automatically when the time is transformed between nodes.

4.3.4. Space Model

It is not enough to specify the time of the phenomenon in a distributed CPS, because multiple components of the CPS may observe the same phenomenon from different positions. Consequently, the position is also relevant to the processing of information. In currently existing systems multiple position data formats are used. They reach from geodesic coordinates as used by GPS [84] to Cartesian projections such as Universal Transverse Mercator (UTM) [77] to abstract representations such as topological maps. Each of these representations are optimized towards a special application. However, each sensor used to acquire positional information typically only delivers a single representation, which is optimized towards its general usage scenario. GPS-Sensors for example often output their information in *National Marine Electronics Association (NMEA)* format. This format is very well suited for navigation on streets, but is to coarse to be used indoors. This format can be transformed to UTM format using mathematical operations.

The information may also be transformed to topological representations such as street maps. The opposite is generally more difficult and produces statically unknown uncertainties depending on the current information. As a consequence, this thesis considers position to be a multi-dimensional uncertain value with a special anchor point providing an origin. The individual positional information can be transformed from one representation to the other if a mathematical transformation description is available. This transformation creates additional uncertainty in the data similar to the time transformations described in Section 4.3.3. An example is the transformation of the RFID-based detection of an entity entering a room. This information is a topological position. If the GPS-coordinates of the room are known, the topological information can be transformed to NMEA information. However, the current position of the entity is uncertain within the room. This introduces additional uncertainty in relation to the size of the room. ACTrESS, as described in Section 3.6, provides a similar system enabling a generalized position information system for ubiquitous architectures. However, this system uses a root context as base to a hierarchical tree of position representations, which is optimized towards minimum information loss. For a general purpose system this approach is not suited since existing positional representations might not be transformable at all. For example, two topological maps without common nodes are inherently incompatible. However, if reasoning and information transfer only happens between components of each topological map, no information is transfered between the disconnected sub-graphs of the maps and the system can process the information separately in each sub-system.

This thesis proposes a local position domain approach similar to the local time domain approach described in Section 4.3.3. Each component of the distributed CPS has its own positional representation that is transformed to another representation if the information is transferred to another component. This enables a very efficient usage of local sensor equipment, but still provides the ability to use local sensors on remote components with increased uncertainty. Additionally, all representation may be used to filter information regarding position. Similar to the time domain, the position context need to be considered as a region, which the event relates to. The region is expressed using the possibly multi-dimensional uncertainties, which are attached to the certain center of the region. The layout of the region is dependent on the used coordinate representation. For Cartesian coordinates the uncertainty description only allows the specification of rectangular areas, whereas for polar coordinate the regions always indicate circle or sphere segments.

4.3.5. Producer ID

The *Producer ID* uses unique identifiers of nodes in the system to enable an identification of the source of the information. It enables a detection of loops in the processing. These loops are typically unwanted, as they produce an infinite amount of output events, which may have a decreasing uncertainty. This data incest is highly dangerous to the system, as it creates invalid context information passed to other components. The *Producer ID* detects such loops and eliminates them by preventing the creation of the invalid information and also by preventing the additional overhead of producing the events. The producer ID is an uncertainty free attribute, as the producer of the data attaches its own id to the created event, which is always an exact information. The ID supports a special combination operation $*$ used in transformations to update the producer of the information and prevent loops between brokers. This operations follows the same definition as the *Event ID* explained in Section 4.4.7. Additionally, it supports the generic discrete value comparison operations.

4.4. Abstract Sensor Event Model

The described context information regarding time, space producer and uncertainty needs to be combined with the various sensor data to a single entity used in the system. To this end, ASEIA uses the abstraction proposed by Marzullo in 1989 [108] as inspiration. Marzullo proposed the abstraction of actual sensor through a concept called 'Abstract Sensors', which hides the actual implementation of the sensor and replaces it with a more general concept. This enables the designer of the system to limit the specification of the sensor to the information relevant for the actual application. As a consequence, the system could use any sensor which fulfills the specified requirements. Additionally, the actual implementation itself may be composed of multiple sensors to enhance the performance for example, an abstract distance sensor may be implemented using three redundant infrared distance sensors.

This thesis extends the concept from specification in the design phase to a machine readable description of these parameters at run-time. In distributed systems like WSN communication parameters such as endianess and latency are relevant too. As a consequence the abstraction is generalized to handle communication parameters, format information, context and semantic information encapsulated in ASEs. These events may be transmitted, processed and evaluated by applications distributed in the network.

The first challenge is the specification of the necessary contained information in ASEs. The context is described in Section 2.1.3 and contains at least information on time, space and quality. The semantic information is needed to enable automatic selection of applicable transformations, based on ontological information. The decision on applicable transformations is typically done when the actual event channel between publisher and subscriber is established. During the life-time the selected transformations do not change. This enables a separation of dynamic and static context of the event. Consequently, the ASE specification is separated into two parts: on the one hand the *Event Type* containing the semantic data endianess, data format, scale and reference data and on the other hand the *ASE* containing the actual phenomenons data and the data on the context attributes. This separation provides the benefit of efficient transmission of events, since static information does not need to be transmitted periodically, This decreases network load. The benefit of this optimization largely depends on the creation and destruction rate of channels.

The encapsulation of all sensor information in ASE enables a homogeneous design of complex processing systems even for heterogeneous information sources. Additionally, the systems transmission and processing may be optimized to enable resource-efficiency by reusing established transformations. Finally, the semantic annotation and the context provide additional information to extend the adaptability of the CPS applications towards changes in the environment or the systems structure.

The general event structure of ASE is a tuple of attributes. Each attribute represents a singular parameter of a phenomenon or its context. It includes information on value, type, scale, unit, uncertainty and semantics of the contained information. The separation of the generic ASE in a static part and a dynamic part is done through the separation of the attributes which the ASE consists of. The static information of the attribute consists of data type, scale, unit and semantic data. The dynamic information is formed by the value and the uncertainty data. Figure 4.5 shows the generic structure of an ASE as an Unified Modelling Language (UML) class diagram.

Each attribute is identified by a unique *Attribute ID* which identifies the semantics of the attribute. The set of *Attribute IDs* forms the basic vocabulary of the semantic data of the

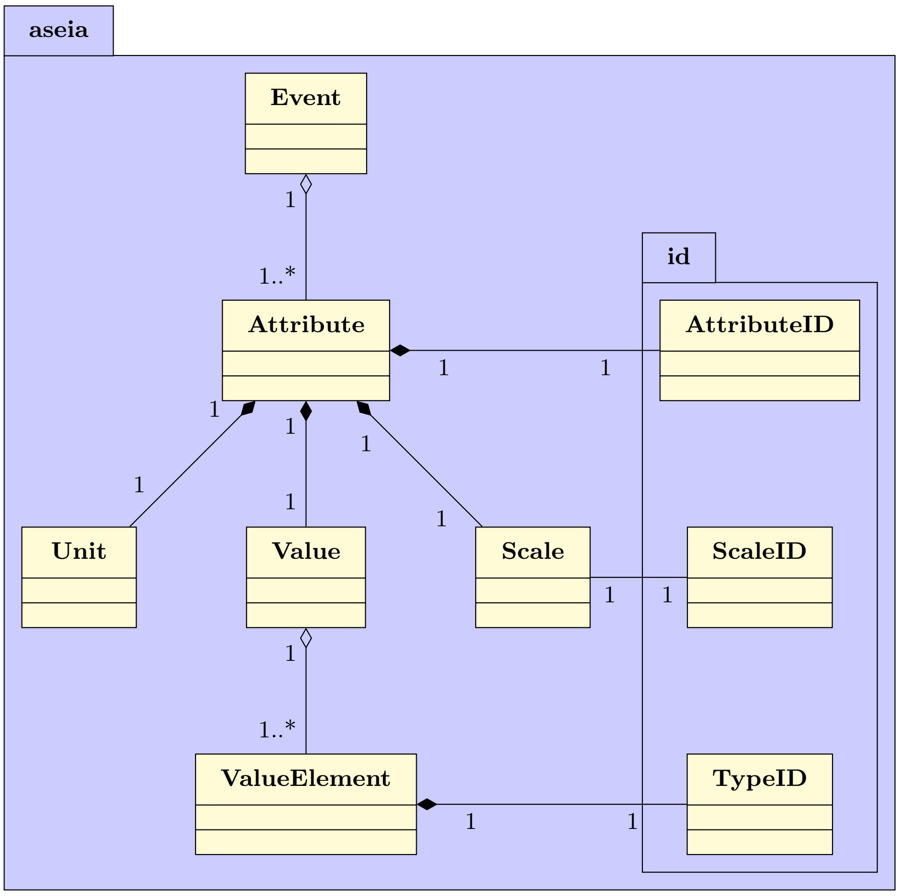


Figure 4.5.: UML class diagram showing the general structure of an ASEIA Event.

system. To form an attribute the attribute id $a^{id} \in \mathbb{N}$ is combined with a value $a^v \in \mathcal{V}$, a unit $a^u \in \mathcal{U}$ and a scale $a^s \in \mathcal{S}$ as shown by Equation 4.19.

$$a = (a^{id}, a^v, a^u, a^s) \in \mathcal{A} = \mathbb{N} \times \mathcal{V} \times \mathcal{U} \times \mathcal{S} \tag{4.19}$$

Each *Attribute* has an associated *Attribute Type* ${}^{T}a$, which consists of all information besides the actual value and uncertainty. The *Attribute Type* is considered as the static information of an *Attribute* that is used to establish channels between publishers and subscribers. The data exchanged in the channel consists of value and uncertainty and is called ${}^{V}a$, as shown in Equation 4.20 and 4.21

$${}^{T}a = (a^{id}, {}^{T}a^v, a^u, a^s) \tag{4.20}$$
$${}^{V}a = (a^v) \tag{4.21}$$

The set of all ASEs $\mathcal{E}$ form a subset of all possible combinations of *Attributes*. However, for one of these combinations to be a valid ASE some requirements need to be fulfilled. The combination is only allowed to contain any *Attribute ID* at most once and the combination needs to contain at least the mandatory context attributes $\mathcal{B}$ as defined in Section 4.3. Equations 4.22 and 4.23 show these properties of ASEs.

$$\mathcal{E} \subset P(\mathcal{A})/\left(P(\mathcal{B})/\mathcal{B}\right) \tag{4.22}$$

$$\mathcal{E} = \left\{(a_0 \dots a_n)|\forall i, j \in [0, n] : i \neq j \rightarrow a_i^{id} \neq a_j^{id}\right\} \tag{4.23}$$

An *Event Type* is composed of all contained *Attribute Types* and is called ${}^{T}e$. Consequently the ASE value is formed by all contained *Attribute Values* and is called ${}^{V}e$, as described by Equations 4.24 and 4.25.

$${}^{T}e = \left({}^{T}a_0, \dots, {}^{T}a_n\right) \tag{4.24}$$

$${}^{V}e = \left({}^{V}a_0, \dots, {}^{V}a_n\right) \tag{4.25}$$

The following sections describe the representation of the values, units, scales, operations and semantic data. The representation of uncertainty is described in Section 4.3, which describes the context attributes mandatory for an ASE.

4.4.1. Attribute Values

Attribute Values store the dynamic data of the attribute. *Attribute Value Types* store the static information on the data format of a specific *Attributess*' values. The values represent matrices of $n \times m : n, m \in \mathbb{N}^0$ instances of a numerical data type T. Each type T has a unique id $id_T \in \mathbb{N}$ associated to it. Additionally, it may include uncertainty information indicated by the uncertainty parameter $u \in \{0, 1\}$. The value and the uncertainty of an attribute are strongly correlated. This thesis uses uncertainty described as intervals which is described in Section 4.3.2.

Independent of the chosen type and uncertainty *Attribute Values* support the generic operations defined for a linear algebra with an inner product on vector space $V^{n\times m}$ over a field K. These operations are addition $+ : \left(V^{n\times m}\right)^2 \rightarrow V^{n\times m}$, scalar multiplication $\cdot : K \times V^{n\times m} \rightarrow V^{n\times m}$, inner product $<>: \left(V^{n\times m}\right)^2 \rightarrow K$ and matrix multiplication $\cdot : \left(V^{n\times m}\right)^2 \rightarrow V^{n\times k}$. Additionally to the basic operations, the attribute values support element-wise operations like boolean comparisons and non-linear functions.

The value type $vt \in \mathcal{VT}$ reflects the structure of the value of an attribute. It can be represented as a tuple of type id $id \in \mathbb{N}$, rows $r \in \mathbb{N}$, columns $c \in \mathbb{N}$ and uncertainty indicator $u \in \{0, 1\}$, as shown in Equation 4.26:

$$(id, r, c, u) \in \mathcal{VT} = \mathbb{N}^3 \times \{0, 1\} \tag{4.26}$$

Value types cannot be modified and only allow comparison regarding equality $\equiv$ and inequality $\not\equiv$.

4.4.2. Attribute Units

Similar to the Boost Unit Library [140], the unit of the attribute is represented as combination of the basic SI-Units [82]. The SI-System defines seven basic units that are combined using multiplication and exponents. Additionally, the definition of radians and steradians are meaningful for physical processes. The unit of force Newton can be represented as shown in Equation 4.27.

$$[F] = \mathrm{N} = \frac{\mathrm{kgm}}{\mathrm{s}^2} = \mathrm{sr}^0 \cdot \mathrm{rad}^0 \cdot \mathrm{cd}^0 \cdot \mathrm{mol}^0 \cdot \mathrm{K}^0 \cdot \mathrm{A}^0 \cdot \mathrm{s}^{-2} \cdot \mathrm{kg}^1 \cdot \mathrm{m}^1 \tag{4.27}$$

Consequently, ASEIA stores and uses units $a^u \in \mathcal{U}$ as nine tuples containing the exponents as real values $exp \in \mathbb{Q}$ as shown by Equation 4.28.

$$a^u \in \mathcal{U} = \mathbb{Q}^9 \tag{4.28}$$

Addition and subtraction of attributes is only possible if their units match. Multiplication and division of attributes produces new attributes with different units as output, since the base units are multiplied element-wise. The unit system forms an Abelian Group $(\mathcal{U}, \cdot)$ with an identity element. This element is the *Dimensionless* unit $(0, 0, 0, 0, 0, 0, 0, 0, 0)$. Similar to other type information used, ASEIA units only support comparison regarding equality $\equiv$ and inequality $\not\equiv$.

4.4.3. Scale Representation

The SI-System [82] and the Boost Unit Library [140] define the scale of the value to be an inherent part of the unit. However, ASEIA separates the scale into an independent parameter, as the considered linear scaling which uses a factor in the form of 10^x is too specialized to be used in general purpose physical processes. Physics and engineering also use other scales such as *Dbm* or *Db* for processes and phenomena with special characteristics. ASEIA introduces a scale system by using an object-oriented approach. The scale a^s is separated into a strictly monotonous function f defined on the whole domain of observations of the phenomenon and a reference point r. The value attribute of the attribute is then constructed from the observation o according to Equation 4.29. Since the function f is strictly monotonous and needs to be defined on the whole domain of o, an inverse function f^{-1} always exists.

$$a^v = f_{a^s}(o, r_{a^s}) \tag{4.29}$$

As a result, the scale can be omitted for value computations as long as function and reference are the same. If they differ, a scale change needs to be executed to translate an attributes' value a^v_0 to the domain of another attribute a^v_1. This translation is done by applying the scaling functions according to Equation 4.30

$$a^v_1 = f^{-1}_{a^s_1}(f_{a^s_0}(a^v_0, r_{a^s_0}), r_{a^s_1}) \tag{4.30}$$

An example is the translation of a temperature from Fahrenheit to Celsius.

$$\begin{aligned}
f_F(o,r) &= {}^{9}\!/_{5} \cdot o + r\\
f_F^{-1}(v,r) &= {}^{5}\!/_{9}\,(v_F - r)\\
r_F &= -459.67\\
f_C(o,r) &= o + r\\
f_C^{-1}(v,r) &= v - r\\
r_C &= -273.15\\
v_C &= f_C(f_F^{-1}(v_F, r_F), r_C)\\
&= {}^{5}\!/_{9}\,(v_F + 459.67) - 273.15\\
&= {}^{5}\!/_{9}\,(v_F - 32)
\end{aligned}$$

Another example is the translation of the phenomenon power from Dbm to mW.

$$\begin{aligned}
f_{dbm}(o,r) &= 10 \log \frac{o}{r}\\
f_{dbm}^{-1}(v,r) &= 10^{v/10} r\\
r_{dbm} &= 1m = 0.001\\
f_{mW}(o,r) &= 1000 \cdot o + r\\
f_{mW}^{-1}(v,r) &= \frac{v-r}{1000}\\
r_{mW} &= 0\\
v_{mW} &= f_{mW}(f_{dbm}^{-1}(v_{dbm}, r_{dbm}), r_{mW})\\
&= 1000 \cdot 10^{v_{dbm}/10} 0.001 + 0\\
&= 10^{v_{dbm}/10}
\end{aligned}$$

4.4.4. Attribute Operations

Attributes allow multiple operations to use them directly in the CPS-Application. The goal of ASEIA is a seamless integration of the communication facilities into the CPS-Application. Therefore, attributes extracted from received ASEs can be used directly for computations, as they allow a lot of computation operations.

Each attribute supports operations to compare them regarding their value and their possibly existing uncertainty as well as their static information. Attributes directly support the operations defined for the *Attributes' Values*. This enables computations on *Attributes* without additional conversion. In addition to the operations defined on values, attributes do additional checks, such as the comparison of scale and unit. This allows the formulation of type-aware computations that minimize the possibility of programming errors. If scale or data type of the value need to be changed, additional operations are available that change the *Attribute Type*. These operations are rescaling (multiplication with a *Scale*), casting of the numeric type used in the value and resizing of the matrix contained in the value. Comparison of attributes can either be done regarding the attributes' type or regarding the attributes' value. Type comparisons only support equality and inequality tests, but value comparisons enable more types of comparisons based on the value type and the presence of uncertainty. An overview of the supported operations of the different attribute types is shown in Figure 4.6.

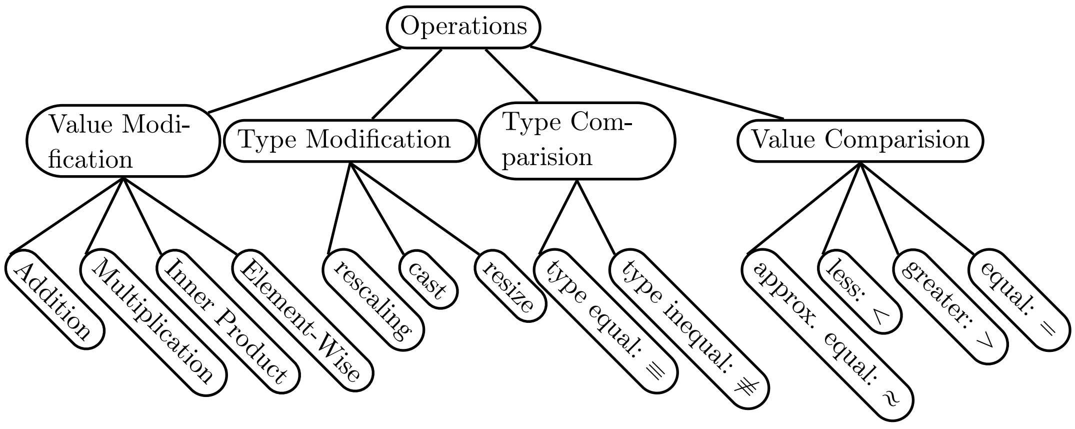


Figure 4.6.: Overview of supported operation classes on *ASE Attributes*.

The type comparison operations $\oslash = (\equiv, \not\equiv)$ combine the comparison operations on the sub-types of the attribute unit, scale, value type and *Attribute ID*, as shown in Equation 4.31-4.33.

$$\equiv, \not\equiv : \mathcal{A} \times \mathcal{A} \to \{0,1\} \tag{4.31}$$

$$^{T}a_0 \equiv^{T} a_1 \doteq \left(a_0^{id} \equiv a_1^{id}\right) \wedge \left({}^{T}a_0^{v} \equiv {}^{T}a_1^{v}\right) \wedge \left(a_0^{u} \equiv a_1^{u}\right) \wedge \left(a_0^{s} \equiv a_1^{s}\right) \tag{4.32}$$

$$^{T}a_0 \not\equiv^{T} a_1 \doteq \neg \left({}^{T}a_0 \equiv {}^{T}a_1\right) \tag{4.33}$$

The value modification operations $\diamond = (=, -, +)$ are directly applied to the values of the attributes. However, only compatible attributes may be used in the process, as described by Equation 4.34. In case invalid attributes are used, the system detects the error and outputs a *void* attribute $\perp$ indicating an error. The multiplication operations check less parameters of the *Attribute Type* because the output of the operations is another *Attribute Type* with possibly modified unit, scale and value dimensions.

$$a_0 \diamond a_1 = \begin{cases} a_0^v \diamond a_1^v & , {}^{T}a_0 \equiv {}^{T}a_1 \\ \perp & , \text{otherwise} \end{cases} \tag{4.34}$$

The attribute operations are available to the CPS-Applications as well as the *Transformations* used to operate on whole ASEs, as described in Section 4.5. The operations defined on whole events are *Transformations* and are described in Section 4.5.

4.4.5. Event Schemes and Event Format

To enable an easy integration of ASEIAs extended functionalities into existing P/S systems, a separation of the *Event Type* into a content description and a data format description is beneficial. The pure content description is realized by the *Event Scheme* ${}^{S}e$, whereas, the data format is described by the *Event Format* ${}^{F}e$ information. The *Event Scheme* is defined as the set of all Attribute Identificator (AID)s contained in the event e, see Equation 4.35.

$$ {}^{S}e = \left\{ a_i^{id} | a_i \in e \right\} \tag{4.35}$$

The *Event Format* is realized by the *Value Type*, *Unit* and *Scale* of all contained *Attributes*, see Equation 4.36.

$$ {}^{F}e = \left\{ ({}^{T}a_i^{v}, a_i^{s}, a_i^{u}) | a_i \in e \right\} \tag{4.36}$$

The *Event Scheme* represents a classification system for the content of events, a similar approach is also used in the *HERMES* middle ware, see Section 3.4. Each *Event Scheme* represents an instance of an Abstract Sensor (AS) which communicates by using ASEs. The *Event Scheme* defines the semantic classification of the existing sensors based on the contained information.

The *Event Format* defines the used data types, scaling system and values as well as units of the attributes and their references. This information is relevant to establish a *Channel* between an arbitrary *Publisher* and an arbitrary *Subscriber* exchanging information according to the content description of the same *Event Scheme*. The *Event Format* is therefore coupled to the *Event Scheme* and extends it to the complete static information of an ASE in form of *Event Types*. However, the independence of *Publisher* and *Subscriber* enables them to use different *Event Formats* for the same content. A classical P/S-Channel can only be established if *Event Scheme* and *Event Format* fit, which limits the automatic communication of independent CPS components. ASEIA solves this problem by using *Transformations*. These are described in Section 4.5 to adapt the *Event Scheme* and/or *Event Format*.

Event Scheme and *Event Format* are basic concepts to establish *ASEIA-Channels* and need to be efficiently represented and processed even by low-power systems. The representation of *Event Schemes* is described in Section 4.4.7.

4.4.6. Event Hierarchy

To ease the definition of different *ASE Schemes* representing numerous heterogeneous AS, an *ASE Hierarchy* is beneficial. This hierarchy specifies the semantic vocabulary of the system it is used in and enables reuse of existing definitions of *ASE Schemes* by extending them with additional information or combining them.

The hierarchy forms a Directed Acyclic Graph (DAG) with a single root node representing the **Base Event**. It contains the necessary attributes to describe general phenomena in an unknown environment according to the *Context*, see Section 4.3. Elements of the DAG can be extended by inserting additional AIDs or by combining existing elements to form new extended *Event Schemes*. The combination of two *Event Schemes* to a new event scheme containing all information of the two individual *Event Schemes* without duplicating redundant information. The generated *Event Schemes* are more specific and contain more information than basic ones closer to the **Base Event**. The resulting hierarchy is very similar to an object-oriented class hierarchy, see [70], enabling inheritance and automatic conversion of more specialized elements to more general ones.

An example hierarchy for the robotic and vehicle scenario is shown in Figure 4.7. In this picture multiple AS are represented that relate to generic physical phenomena, which may be observed in these scenarios. The **Acceleration** *Event Scheme* contains the necessary information to represent an n-dimensional acceleration value. In addition to the actual value of the acceleration it may contain additional information on the position and time of acquisition of the data based on the definition of the mandatory *Context*. For multi dimensional acceleration values, applications may need the orientation of the sensor to map the individual vector components to their coordinate system. In this case the AS representing the phenomenon **DirectedAcceleration**, which combines the **Acceleration** information with the **Orientation** of the sensor. It can be used to supply the CPS-Application with the necessary information. **DirectedAcceleration** is further away from the root compared to **Acceleration** and **Orientation**, which means that it contains more specialized information. On the other hand, it extends the two *Event Schemes* and may automatically be converted into instances of these, if a CPS-Application only needs **Acceleration** or **Orientation** information. This allows all generated oriented sensor values to be used also as orientation updates of the sensor itself. This follows exactly the behavior of an object-oriented class hierarchy. However, converting a generic *Event Scheme* like **Acceleration** to a more special one such as **DirectedAcceleration** is only possible by using *Transformations*, which are described in Section 4.5.

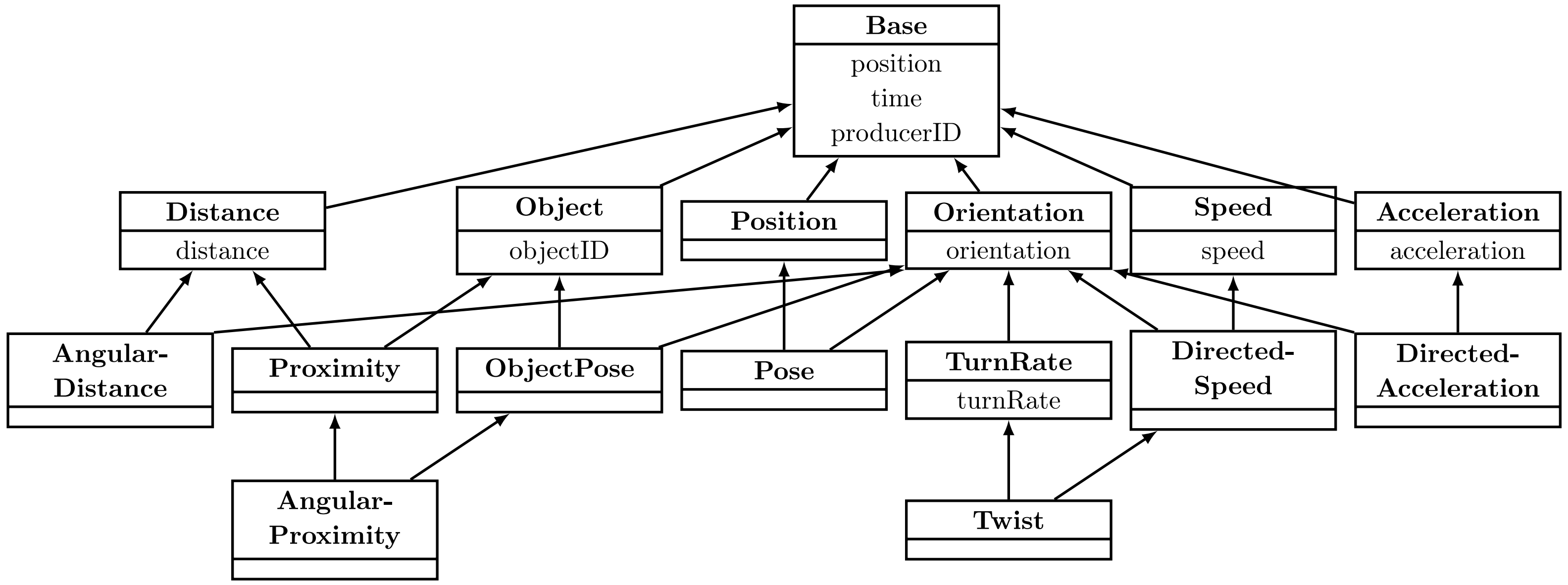


Figure 4.7.: An example event hierarchy usable in the robotic and the vehicle context.

4.4.7. Semantic Annotation - Event IDs

A semantic annotation of the events is necessary to enable a content based reasoning. This reasoning is the baseline of all ontology-based systems enabling the deduction of new information from existing ones. In the case of ASE, the semantic annotation is the foundation of the transformation selection process which is executed on channel creation. The annotation needs to contain the condensed information classifying the AS. However, the annotation needs to be simple enough to be used for low-level systems and still supports necessary operations to check for equality and inheritance between different AS.

As proposed by LooCI, see Section 3.3.3, the semantic annotation used in the thesis will not directly create an ontology, but rather establish a vocabulary of ASE types with the according inheritance relations and conversion through the *Transformations.* The content of the vocabulary needs to be extensible and still provides backwards compatibility to enable an extension of the system on run-time.

The thesis uses Event Identificator (EID)s that are based on one dimensional integer values, which are dynamically computed from the *Event Scheme.* The vocabulary grasps the content of the ASEs and not the format or meta-information. The attributes, which are identifiable by their unique AID, exactly define the content of the event. Therefore, they provide a reasonable information source to generate EIDs. Since the amount of attributes in an ASE might be potentially large, a compression function is necessary to combine multiple AIDs to a single integer value.

Preuveneers and Berbers [126] proposed and proved a combination system that is based on prime numbers. The system efficiently computes unique ids for the event tree by using a list of primes P and sequence numbers of event types from N. Each added event type is assigned a sequence number $n_i \in N$ and a prime number $p_i \in P$. The root of the tree is the base event $EVENT$ and is always assigned $p_0 = 1$[1] and $n_0 = 1$. The id of the base event b_0 is defined as $b_0 = 1$. The resulting id is computed as product of all parents' ids b_j and the assigned prime number $b_i = b_j \cdot p_i; i > j; i, j \in N$. An inheritance test is performed by the predicate $b_0 \bmod b_1 = 0$.

This approach has two major drawbacks. Firstly, the inheritance scheme is strictly single inheritance as it only allows the definition of new event schemes based on existing ones. Combinations of existing event schemes are not supported. The second drawback is the assignment of the prime numbers based on event scheme creation. The necessary global counter for prime number enumeration is very hard to realize for distributed systems.

The thesis proposes a modified version of their approach resulting in EIDs e^{id}, which are a subset of the natural numbers, see Equation 4.37. The foundation of the EID generation are prime numbers, which are assigned to attributes based on the attributes' unique IDs a^{id}. The event scheme ID e^{id} is generated by multiplying the prime numbers of the contained attributes, as described by Equation 4.39. The event ID 0 indicates a special id used in filters that specify any event schemes e^{id}_{any}. The event scheme combination operation is implemented by the construction of the union of the individual *Event Schemes* ${}^{S}e_{combined} = \bigcup {}^{S}e_i$ and the generation of an EID for the resulting scheme, as described by Equation 4.40. Extensions with specific attributes are possible, since a pseudo event scheme can be constructed for a single attribute ${}^{S}e_a = a^{id}$. The definition by using the AID

[1]even though 1 is not a prime number itself

set automatically filters out the common root of all the combined events and guarantees uniqueness of the resulting EIDs.

The inheritance check $a > b \iff a \bmod b = 0$ works exactly as described by Preuveneers and Berbers by using the modulo operation to check for remainders, see Equation 4.43. This approach automatically enables testing of inclusion of a certain attribute in an event scheme solely based on the EID. Since event schemes are sets of attributes, this can also be used to test for the inclusion of arbitrary AID sets by creating a pseudo event.

The drawback of the extended system is the increase in the value of the ids and a much sparser EID space. The created ids grow very quickly and need a careful decision on the used data type to represent the EID. A large data type increases the size of each ASE transmitted through the network, whereas, a small data type is vulnerable to integer overflow disabling the extension of the system with extended *Event Schemes* or *Attributes*.

$$e^{id} \in \mathbb{N} \tag{4.37}$$

$$prime(n) = \mathbb{N} \rightarrow \mathbb{P} : \text{n-th prime number} \tag{4.38}$$

$$e^{id} = \prod_{a_i^{id} \in {}^{S}e} prime(a_i^{id}) \tag{4.39}$$

$$e^{id}_{combined} = \prod_{i=0}^{n} e_i^{id} = \prod_{a_j^{id} \in \bigcup_{i=0}^{n} {}^{S}e_i} prime(a_j^{id}) \tag{4.40}$$

Event ID Uniqueness - : *For each pair of events $e_0, e_1 \in \mathcal{E}$ a difference in Event Schemes yields a difference in EIDs:*

$$\forall e_0, e_1 \in \mathcal{E},\ {}^{S}e_0 \neq {}^{S}e_1 \iff e_0^{id} \neq e_1^{id} \tag{4.41}$$

Proof. Consider two different *Event Schemes* ${}^{S}e_0$ and ${}^{S}e_1$ yielding the same EID ${}^{S}e^{id}$. By applying the EID generation Equation 4.39, the event schemes can be reduced to *Attribute ID* sets, which need to be different, because each attribute has a unique AID. As consequence, the product of the primes associated with these *Event Schemes* needs to be equal, see Equation 4.42.

$$\prod_{a_i^{id} \in {}^{S}e_0} prime(a_i^{id}) = \prod_{a_i^{id} \in {}^{S}e_1} prime(a_i^{id}) \tag{4.42}$$

However, this is in conflict with the **fundamental theorem of arithmetic**, which states that any natural number can be represented by exactly one product of prime numbers [79]. Therefore, the original assumption of the equality of the EIDs is false.

□

Event Inheritance - : *For each pair of* Event Schemes ${}^{S}e_0$, ${}^{S}e_1 \in {}^{S}\mathcal{E}$ *iff* ${}^{S}e_1$ *inherits from* ${}^{S}e_0 \iff e_0^{id} \in e_1^{id}$*:*

$${}^{S}e_1 \supset {}^{S}e_0 \iff e_1^{id} \bmod e_0^{id} = 0 \tag{4.43}$$

Proof. Consider e_0 and e_1 to have the same *Event Scheme*. It follows based on Equation 4.41 that the EIDs need to be the same as well $e_0^{id} = e_1^{id} = e^{id}$. Applying the test stated in Equation 4.43 yields $e^{id} \bmod e^{id} = 0 \rightarrow e_0 \in e_1$. If ${}^{S}e_0$ and ${}^{S}e_1$ have different *Event Schemes*, two cases may occur:

1. ${}^{S}e_1 \supset {}^{S}e_0$ and $e_0^{id} \in e_1^{id}$
2. ${}^{S}e_1 \not\supseteq {}^{S}e_0$ and $e_0^{id} \notin e_1^{id}$

Case 1 means that e_1^{id} can be separated into e_0^{id} and the id of the extension set e_{ext}^{id}:

$$e_1^{id} = e_0^{id} \cdot \prod_{a_i^{id} \in {}^{S}e_1 / {}^{S}e_0} a_i^{id} = e_0^{id} \cdot e_{ext}^{id} \tag{4.44}$$

$$e_0^{id} \cdot e_{ext}^{id} \bmod e_0^{id} = e_{ext}^{id}(e_0^{id} \bmod e_0^{id}) = 0 \tag{4.45}$$

$$\tag{4.46}$$

Following Equation 4.43 the inheritance test is simplified to Equation 4.45, which proofs $e_0 \in e_1$.

Case 2 means that ${}^{S}e_1$ and ${}^{S}e_0$ can each be separated in a common *Event Scheme* ${}^{S}e_{common} = {}^{S}e_0 \cap {}^{S}e_1$ and two non-empty remainder *Event Schemes* ${}^{S}e_{1'} = {}^{S}e_1 / {}^{S}e_{common}$ and ${}^{S}e_{0'} = {}^{S}e_0 / {}^{S}e_{common}$, which are pair-wise disjunct. The inheritance test, see Equation 4.47 can be simplified to Equation 4.48 through the application of the rule $ca \equiv cb \bmod \left(\frac{m}{gcd(c,m)}\right)$, with $c = e_{common}^{id}$.

$$e_{1'}^{id} e_{common}^{id} \bmod e_{0'}^{id} e_{common}^{id} = 0 \tag{4.47}$$

$$e_{1'}^{id} \bmod e_{0'}^{id} = 0 \tag{4.48}$$

4.48 cannot be true because ${}^{S}e_{1'} \cap {}^{S}e_{0'} = \varnothing$ and therefore the $gcd(e_{1'}^{id}, e_{0'}^{id})$ is $= 1$.

□

4.4.8. Format Hashes

Similar to the EID, a compact representation of the *Event Format* is necessary to allow efficient checking of ASE compatibility between publisher and subscriber. Therefore, a function is necessary to map the potentially large format definition to single dimensional value. In contrast to *Event Schemes*, no inheritance information needs to be preserved. Consequently, only equality and inequality comparison need to be supported.

The best choice for this mapping is a hash function, that maps the arbitrary long information contained in the *Event Format* to an integer value of fixed size. The function needs to be executed. Therefore, it needs to be executed fast. To prevent mismatches of *Event Formats*, the hash function should minimize collisions. If collisions occur, additional ASE are transmitted through the network increasing the network load. However, no erroneous ASE will be received as these wrongly delivered ASEs are additionally checked regarding network meta-information such as length of serialized data.

The actual choice of hash function is irrelevant as long as the described requirements are met.

4.5. Abstract Sensor Event Transformations

The basic concept of ASEIAs dynamic composition of CPS components are *Transformations* of ASEs flowing through the network to couple existing publisher in the network to subscribers of the CPS applications. These *Transformations* need to be described in a machine readable way to enable the network to deploy them on run-time if necessary. The description consists of the *Event Type* of the input and output events. The system decides whether the execution of the transformation is necessary based on the existing subscriptions and announcements of publishers and subscribers. The *Transformations* consist of the input and output *Event Type* ${}^{T}e_i$, additional *Filter Expressions* $p_i :^T e_i^n \rightarrow 0, 1$ and a function $f : {}^{T}e_i^n \rightarrow {}^{T}e_{out}$ which takes n input events of types ${}^{T}e_i$ and outputs a single event e_{out} of type ${}^{T}e_{out}$. The *Transformation* may be selected according to its specified *Event Scheme ID* and *Event Format*. This generic setup of *Transformations* is displayed in Figure 4.8. On execution of the *Transformation*, the *Filter Predicates* p_i are checked and on positive evaluation of all of them, the *Transformation Function* f is executed.

Transformations are very flexible regarding their parameters. To this end, different types of *Transformations* need to be considered to enable detection of incoming events, processing them and output appropriate events as expected by the subscribers. The different *Transformations* can be classified into *Unary-Transformations* and *N-Ary-Transformations* based on the amount of input events. Additionally, the transformations can be separated based on the *Event Scheme IDs* of the input and output. *Transformations* which contain compatible *Event Schemes* in their set of input event scheme set and their output event scheme set are called *homogeneous*. *Heterogeneous Transformations* on the other hand have disjunct sets of input and output *Event Schemes*. A third classification is based on the abstraction level the *Transformation* acts on. *Attribute Transformations* transform single attributes of incoming events, whereas, *Event Transformations* work on the event as a whole. The following terminology is used in the thesis to describe the different types of *Transformations*.

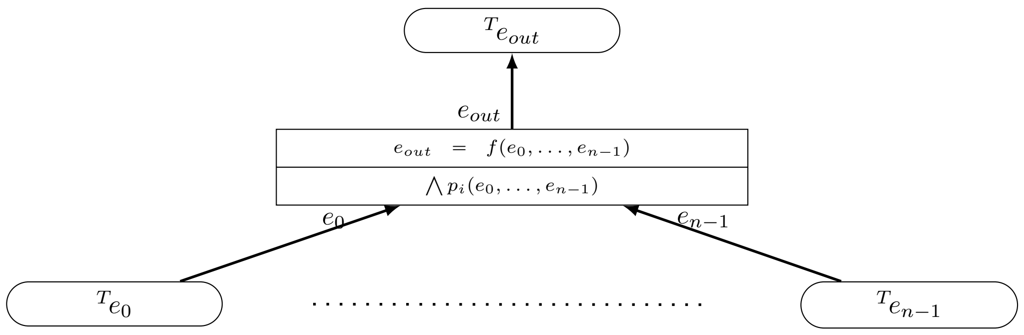

${}^{T}e_{out}$
e_{out}
$e_{out} = f(e_0, \ldots, e_{n-1})$
$\bigwedge p_i(e_0, \ldots, e_{n-1})$
e_0
e_{n-1}
${}^{T}e_0$
${}^{T}e_{n-1}$

Figure 4.8.: Graphical illustration of a generic *Sensor Event Transformation* with n inputs, m filter expressions $p_0, \ldots, p_{m-1}$ and a transformation operation f.

Unary Transformations take a single event of a known *Event Scheme* as input and output a single event of an also known *Event Scheme*. This type of transformation can directly be executed whenever an appropriate event is received by the broker node.

N-Ary Transformations take multiple events of possibly heterogeneous *Event Scheme*s and combine them to a single output event. The incoming events need to be buffered in the node executing the *Transformation*, since the events are typically not received at the same time. The buffering of the events can follow different strategies based on the goal of the application. This can be expressed by buffering strategies such as the ones described in Section 4.6.4.

Homogeneus Transformations output events of the same *Event Scheme* which they also take as input events. This may produce transformation cycles, where the same transformation is executed on the same events repeatedly. This is dangerous, because it will create an infinite amount of events flooding the network. This can only be avoided by appropriate filter expression attached to the transformation.

Heterogeneus Transformations output events of *Event Schemes* which are not part of the input event set. This will never produce transformation cycles producing infinite amounts of events.

Generic Transformations are *Transformations* that modify their behavior according to information provided by the publisher and subscribers on channel creation, which consists of *Event Types* and *Filter Predicates*.

Composed Transformations are *Generic Transformations* created by combining multiple *Transformations* selected from the *Knowledge Base* and placed in a single connected *Transformation DAG*.

Attribute Transformation operate on attributes of events. They can be and be combined with other *Attribute Transformation*s as long as they operate on disjunct attributes of the same *Event Scheme* to form a single *Heterogeneous Event Transformation*. They are are *Generic Transformations* as they need to adapt to the *Event Format* specified by subscriber and publisher.

Event Transformations operate on all attributes of a single *Event Scheme*. The *Event Scheme* describes the necessary attributes for the given transformation. Because of the used *Event Hierarchy*, they may also use events of *Event Schemes* further down the hierarchy.

The classes of the different *Transformations* form a hierarchy, which is displayed in Figure 4.9. Additionally, some example *Transformations* are inserted into the Figure to display the relation of typical operations used to processing sensor data mechanism to the classes of *Transformations*.

The individual *Transformations* are selected as necessary by the brokers in the network and form *Transformation DAGs* in each broker, which are executed on ASE reception. The individual *Transformation* classes and exemplary instances of them are described in the following sections. The creation and execution of the *Transformation DAGs* is described in Section 4.6.

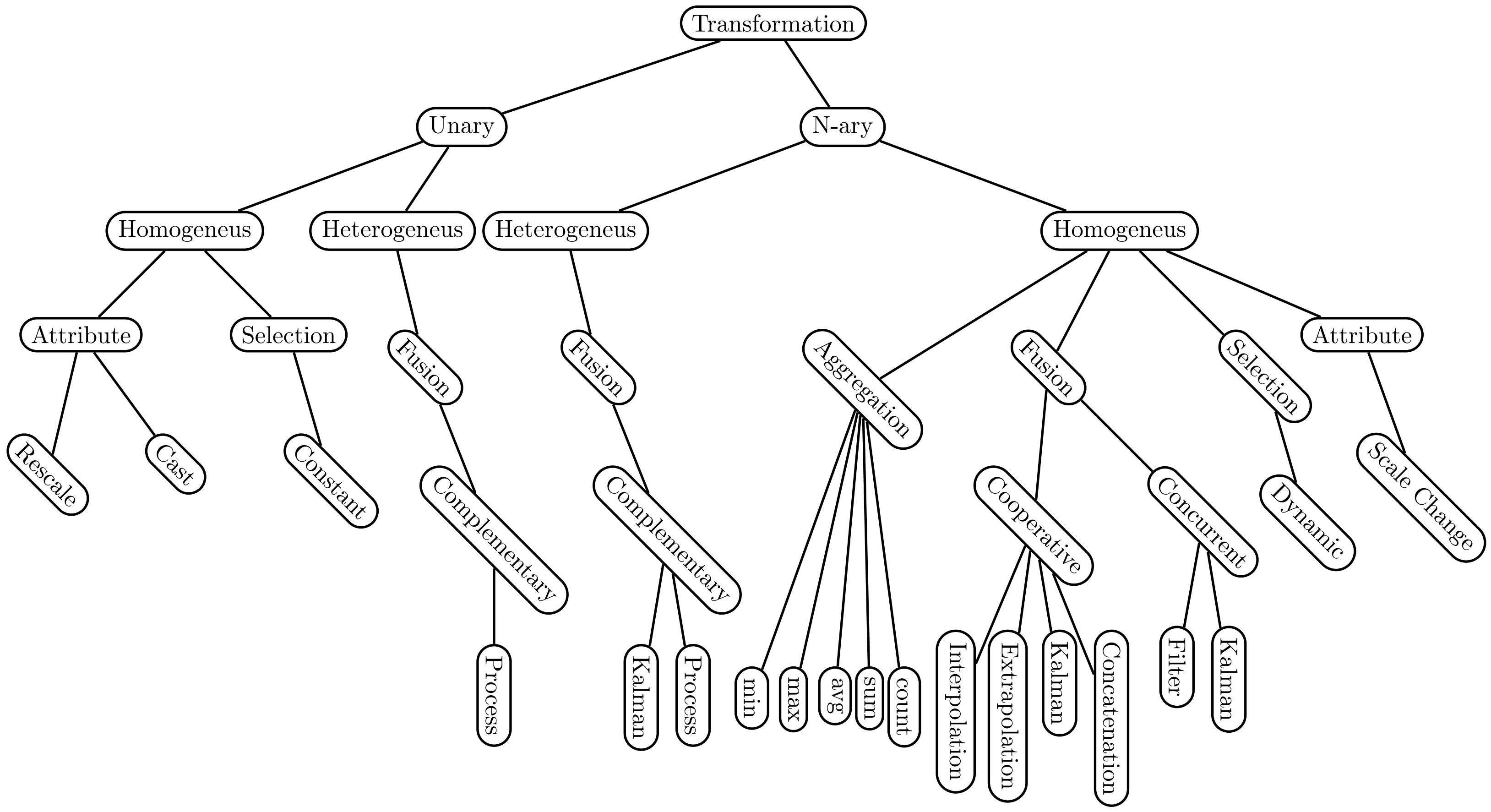


Figure 4.9.: Hierarchy of Event Transformations

4.5.1. Selection

Selection is a major mechanism used in many CEP and CED systems, see Section 3.4 to fulfill functionalities of the *Smart Connection Layer*, see Section 2.2.1. It enables the individual nodes to specify their interest in certain information contained in an event or an *Event Combination*. ASEIA considers selection to happen before any *Transformation* is executed. The selection is executed on three layers of information: *Event Scheme*, *Event Format* and event content. The scheme and format are static information and part of the established channel. Therefore, this selection is executed automatically for any communication in the P/S system. The filtering of *Event Combinations* based on content is executed in the form of special *Selection Transformations*. These consist of a set of filter predicates p_i on a set of event types ${}^{T}E = \left\{{}^{T}e_0, \ldots, {}^{T}e_{n-1}\right\}$. Each predicate consists of a comparison operation, as described in Section 4.3.2, between an attribute of an input event and a constant or the same attribute of a different input event. Consequently, each predicate can be considered to be a tuple of the constant predicate set $P_c = \left\{(i, a_j^{id}, \diamond, c) | 0 \leqslant i < n, a_j^{id} \in \S e_i, {}^{T}c \equiv {}^{T}a_j\right\}$ or the dynamic predicate set $P_c = \left\{(i, a_j^{id}, \diamond, k) | 0 \leqslant i, k < n, a_j^{id} \in {}^{S}e_i, a_j^{id} \in {}^{S}e_k\right\}$ on the input events. These predicates are issued by the subscriber and forwarded to the brokers on channel creation. The brokers include these predicates in the form of *Selection Transformations* in their *Transformation Graphs* to enable native filtering of events. The resulting transformation is always a homogeneous transformation as it forwards one input event on fulfilled predicates.

Some CED systems allow the specification of detection rules allowing the exclusion of an event. ASEIA does not allow this type of operations, as this operations needs the capability of the underlying P/S system to detect if an event does not exists. In asynchronous systems this information cannot reliably be established as the event in question can be delayed arbitrarily. This problem has already been discussed by the authors of *Abstract Events* (Section 3.4). An additional problem is the disambiguation between an event with maximum uncertainty and no event. An event containing maximum uncertainty for each sensor data and context parameter conveys no information and is comparable to receiving no event at all. The approach of event exclusion typically relates to detection events. ASEIA's discrete uncertainty model needs these events to contain a discrete detection sensor data attribute. Instead of specifying that a special detection event did not exist, an event *Filter Expression* can be constructed, which compares the detection event regarding uncertainty. Using this approach similar expressiveness can be achieved, but it also allows the CPS application to detect and react to situations without available information.

The selection mechanism consists of two subtypes that have inherently different properties. The *Static Selection* and *Dynamic Selection* are described in the next paragraphs.

Static Selection The *Static Selection* compares the values of the attribute and the context of the attributes against constant values. Therefore, it may only contain predicates from the constant predicate set. This operation is very efficient, since each predicate only operates on a single event. Therefore, events can be directly classified into "usable" and "unusable" on reception. Consequently, unusable events can directly be discarded without storing them temporarily. The operation of this transformation resembles the "WHERE" statement of SQL expressions and may be used to limit the amount of incoming events, especially in the case of periodic events.

Dynamic Selection The *Dynamic Selection* compares values and context of attributes between incoming ASE. This is a very expensive operation, since in worst case every *Event Combination* of buffered events needs to created and evaluated. This requires many operations to be executed by the buffering filtering system of the transformation. On the other hand, the results directly enable distributed event detection in the CPS. Depending on the used operation in the transformation executed, detection of composed events, abnormal situations and even dangerous situations are possible. As a *Dynamic Selection Transformation* may also contain predicates of the constant predicate set, it can be separated into a purely static part and a dynamic part. An early execution of the static selection might mitigate the performance issues of the *Dynamic Selection.*

4.5.2. Attribute Transformations

The *Attribute Transformations* enable a decoupling of the publisher and subscriber in the data format domain. They only need to agree on the *Event Scheme* as vocabulary to be able to communicate with ASEIA automatically by taking care of the adaption of scale, reference and data types. This allows each component of the CPS to work with the data types best suited to the inherent limitations of the platform which it runs on. An 8-bit micro controller for example can favor 16-bit integers over 32-bits floats to represent the values. Whereas, a general purpose system such as ROS favors floats in SI Base Units because of standardization. Generally, data type and scale are coupled, because depending on the scaling of values different ranges of values are encountered on run-time, which needs different data types to handle them without over- or underflow. However, ASEIA models both *Transformations* as individual transformations that are configured and composed to a single *Event Transformation* if necessary. The following example transformation show the process of *Attribute Transformation* based on an example transformation extracted from the robotic scenario: The robotic scenario shows some deeply embedded distance sensors used to detect the presence of an entity. These sensors may be used by a robot to enhance its positioning if no other absolute position information is present. However, robotic systems typically consider position to be a floating point value either in polar coordinates, as provided by a GPS sensor, or in a Cartesian coordinate system. Typical distance sensors used in the robotic context have a limited range $< 10m$ and seldom surpass millimeter resolution. As a consequence, they provide their data as integer values in centi- or millimeters. To enable the robot to use the distance information a cast from integer to float is necessary together with a rescaling of the value from centimeters to meters. This combined homogeneous unary transformation is shown in Figure 4.10. It consists of two basic *Unary Attribute Transformations*: *Cast Transformation* and *Rescale Transformation*, which are described in the next paragraphs.

Cast Transformation The *Cast Transformation* enables a translation of the *Attributes Value Type*, as described in Section 4.4.1, from one data type to another. It is a *Unary Generic Homogeneous Attribute Transformation.* An example is the translation of a vector or matrix of 32-bit integers to *IEEE754* single precision floats. The *Transformation* only needs a single event as all necessary information is present on creation of the channel through the exchanged *Event Types* between publisher and subscriber. In cases, where the target

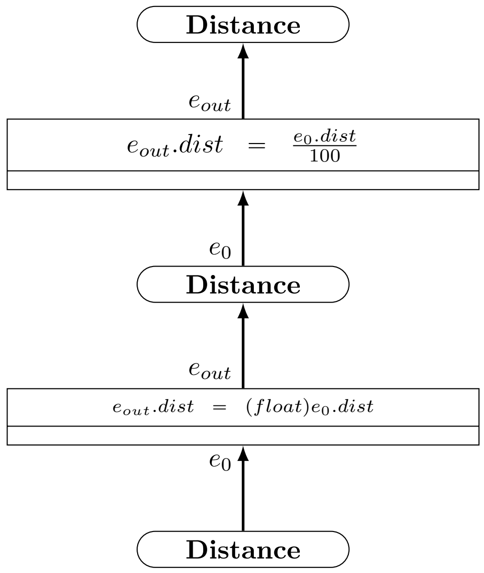

Distance
e_{out}
$e_{out}.dist \;=\; \frac{e_0.dist}{100}$
e_0
Distance
e_{out}
$e_{out}.dist \;=\; (float)e_0.dist$
e_0
Distance

Figure 4.10.: Figure showing the combined transformation used to convert distances from centimeters as integer to meters as float.

data type is larger than the source data type, no additional uncertainty is created. However, in the other case additional uncertainty might be created by the conversion because of a loss of precision for floating point values or because of saturation for integers.

Rescale Transformation The *Rescale Transformation* enables the translation of different linear scales, as described in Section 4.4.3. It enables the rescaling of values between SI-Prefixes like *milli-* or *kilo-*. This behavior is a specialized case of the generic *Scale Transformation*, which is described in the next paragraph. In contrast to the generic one, no additional run time information is necessary to execute the *Transformation*, as all necessary information is contained in the *Event Type*.

Linear scaling as used for SI-Prefixes can be represented by three parameters n, d and r. n and d are the numerator and denominator of the scaling quotient $m = {}^{n}\!/_{d}$ of the linear function $f = mx + r$. r represents the reference point of the scaling. Therefore, the *Event Type* in case of linear scaling directly contains the numerator and denominator and a reference id r_0^{id}. To translate a value v_0 from scale $s_0 = (n_0, d_0, r_0)$ to the scale $s_1 = (n_1, d_1, r_1)$, the difference between reference points $\Delta r_{0,1}$ is needed together with two numerators and denominators, according to Equation 4.49. The *Rescale Translation* can only be used if the reference identified by the two reference ids are equal. Consequently, $\Delta r_{0,1}$ is 0, which simplifies the operation to $v_1 = v_0 \cdot \Delta s_{0,1}$. The value $\Delta s_{0,1}$ is the rescaling coefficient computed on channel creation for each attribute.

$$v_1 = v_0 \frac{d_0}{n_0} \frac{n_1}{d_1} + \Delta r_{0,1} = v_0 \cdot \Delta s_{0,1} + \Delta r_{0,1} \tag{4.49}$$

This *Transformation* only creates additional uncertainty if the multiplication saturates because of overflow or because of a loss of precision for floating point values.

Scale Change Transformations The generic handling of scaling and especially references of coordinate systems is a very important topic in robotics. ROS uses a special subsystem developed by Foote called *tf* [65]. This system enables a hierarchy of coordinate systems to be handled automatically through the specification of the translation and rotation between them. ASEIA provides the *Scale Representation*, see Section 4.4.3, as a generalization of this concept attached to each attribute. The *Scale Change Transformation* is a *Generic Dual Homogeneous Attribute Transformation* enabling a generic translation from one scale to another by providing a functionality similar to *tf transformPose*. To this end, it needs an additional event containing the necessary *Reference Information* for the two scales. This event needs to be provided by the system similar to ROS *tf*. The resulting *Transformation* needs to buffer at least the most recent *Reference Information* to be able to execute. Additionally, the age of the *Reference Information* impacts the uncertainty, which is captured by the time attribute of the transformed event. The time interval is extended to contain the time stamp of the *Reference Information* to enable the CPS Applications to handle the age of the *Sensor Information* as well as the age of *Reference Information*. An example

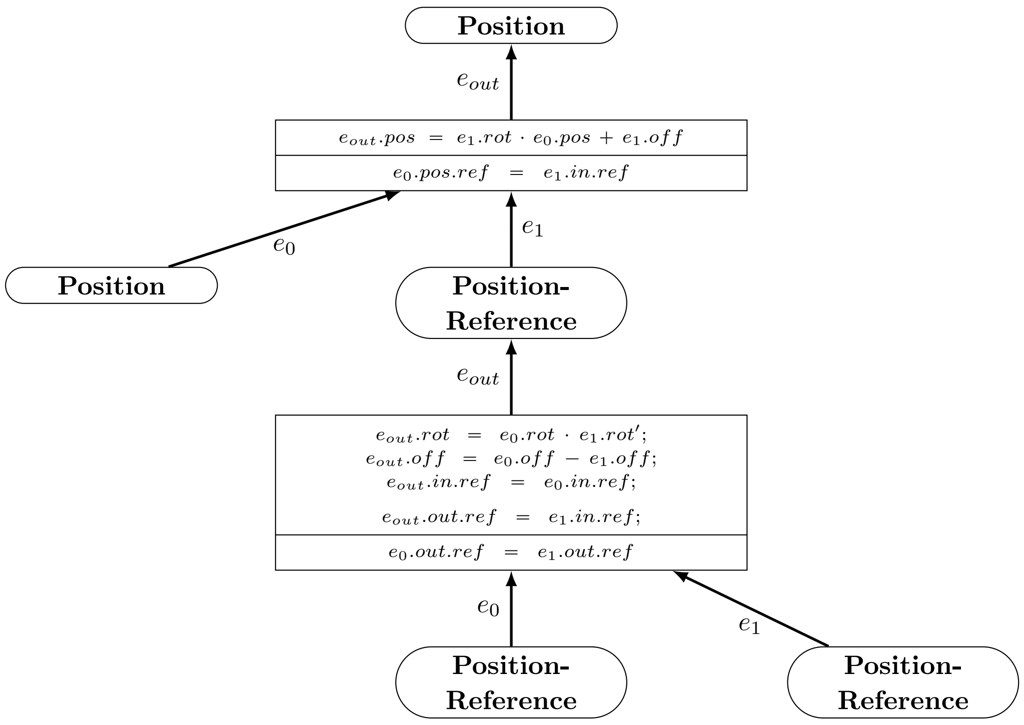

Position
e_{out}
$e_{out}.pos = e_1.rot \cdot e_0.pos + e_1.off$
$e_0.pos.ref = e_1.in.ref$
e_0
e_1
Position
Position-Reference
e_{out}
$e_{out}.rot = e_0.rot \cdot e_1.rot'$;
$e_{out}.off = e_0.off - e_1.off$;
$e_{out}.in.ref = e_0.in.ref$;
$e_{out}.out.ref = e_1.in.ref$;
$e_0.out.ref = e_1.out.ref$
e_0
e_1
Position-Reference
Position-Reference

Figure 4.11.: Figure showing the *Composed Transformation* used to convert **Position** events of one robot translated to another robots coordinate system.

of this transformation is the transformation of *Position Information* of a robot from the robot scenario to another robots coordinate system. This transformation needs additional information to be executed, namely the relation between the two coordinate systems of the robots. The resulting transformation needs to find a common coordinate system of the two robots to deduce the coordinate transformation necessary to transform positions of one coordinate system to the other. The resulting combined transformation is shown in Figure 4.11. This figure shows two *Transformations*. The lower one takes the two *Reference Points* of the two scales of the robots and computes their $\Delta r_{0,1}$. This is based on their relation to a third common coordinate system. This behavior is also the base of *ACTrESS*, discussed in Section 3.6. The upper one uses the resulting $\Delta r_{0,1}$ to change the reference of the incoming positions. For clarification purposes the modification of the time attribute is omitted.

4.5.3. Event Transformations

Attribute Transformations, as described in Section 4.5.2, allow the dynamic adaption of the data formats of publishers and subscribers of the same *Event Scheme* to establish communication between them. *Event Transformations* on the other hand use the semantic annotation to automatically establish channels between publishers and subscribers of different *Event Schemes*. They provide mechanism to execute sensor signal processing within a CPS' **Cyber Layer**, see Section 2.2.3, in the network to dynamically link publishers and subscribers as required by the application. To this end, different types of signal processing might be used. The different classes of processing are described in the next Sections.

4.5.4. Complementary Sensor Fusion Transformations

Complementary Sensor Fusion Transformations are a *Heterogeneous Unary* or *N-Ary Event Transformations*. They use a single or multiple input events $e_{in} = (e_0, \ldots, e_n)$ to compute new sensor events of a different *Event Scheme* $\forall e_i \in e_{in} : e_{out}^{id} \neq e_i^{id}$. The physical process description of the *Transformation* uses the *Attribute Operations*, see Section 4.4.4, to formulate the translation function f of the *Transformation*. Additionally, the *Constant* and *Dynamic Selection Predicates*, see Section 4.5.1, $p_i \in P_C \cap P_D$ may be used to specify filter expressions stating the constraints of the physical process. An example is the transformation of position information of an object to speed information, as shown in Figure 4.12.

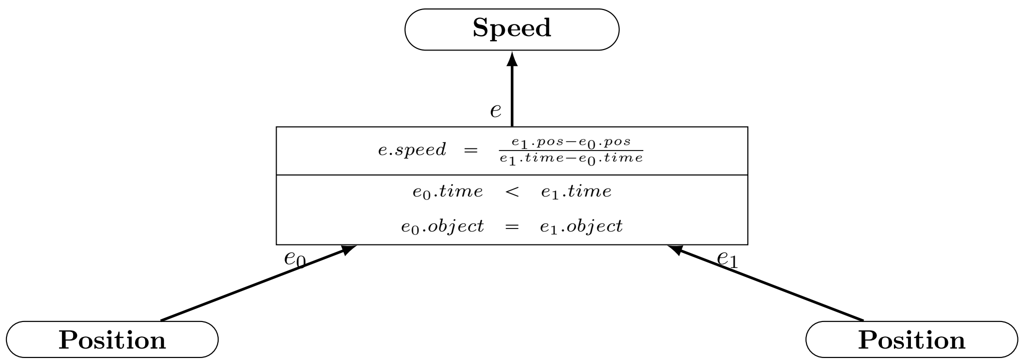


Figure 4.12.: An example heterogeneous transformation to infer speed data from position data.

This *Transformation* uses the difference quotient $\frac{\Delta x}{\Delta t}$ as an approximation of the differential quotient $\frac{\delta x}{\delta t}$. This operation is only valid, if $\Delta t \neq 0$. Therefore, filter expressions are needed to prevent this situation. In this case it is necessary to ensure that the position ASEs relate to the same object and can be ordered regarding time. The ordering is especially important, because the *Continuous Uncertainty Comparison Operations*, see Section 4.3.2, guarantee that ordered intervals are never equal and therefore, the subtraction of these intervals may not contain 0, which ensures the correctness of the physical process description.

This type of *Transformation* can be used to match incompatible *Event IDs* of publisher and subscriber if an appropriate *Transformation* exists. To enable an automatic composition of a CPS, the network needs to be equipped with a database of physical process descriptions suited to the scenario of the CPS. The resulting *Complementary Transformations* are employed dynamically whenever adequate publishers are present in the network and a subscriber for the output *Event Scheme* exists. The following two examples show typical use-cases for *Complementary Transformations* in the robotic and vehicular scenarios.

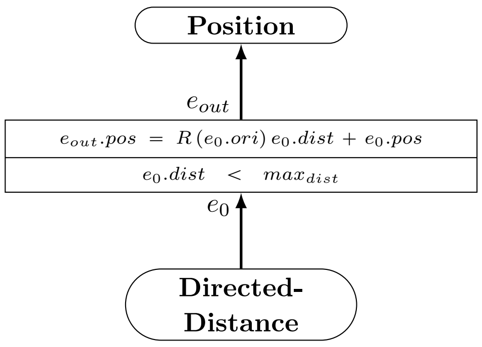


Figure 4.13.: An example heterogeneous transformation to infer an object's position from a distance.

Virtual Speed Sensor The robotic scenario imposes the robots to broadcast their position. However, for path planning it may be necessary to additionally gain speed information for each entity. This might be interesting for robots, but also for indirectly observed humans. A *Virtual Speed Sensor* provides exactly this functionality by differentiating position data over time to speed data. To this end multiple **Position**s need to be observed over a small time scale in either a close area $|e_0.pos - e_1.pos| < \epsilon$ or with the same **Object** Information $e_0.objectID = e_1.objectID$. The resulting transformation is shown in Figure 4.12.

Virtual Object Position Sensor As described for the *Virtual Speed Sensor*, position information is very relevant to the robotic scenario. The existing distance sensors imposed by the scenario can be used to create additional position information to be used directly by the robots or fed to other transformations. To this end, a **DirectedDistance** created by a statically placed sensor can be transformed to a **Position**. This can be done for each distance information that indicates the presence of an object. The **Distance** information $e_0.dist$ needs to be multiplied with an uncertain rotation matrix $R(e_0.ori)$ created by the contained **Orientation** of the sensor. The resulting **Position** $e_{out}.pos$ is related to the **Position** $e_0.pos$ of the original information created by the sensor, which contains the sensor's position. However, no **Object** information may be inferred by this transformation as a distance sensor does not provide the necessary data. Figure 4.13 shows the resulting transformation.

4.5.5. Cooperative Sensor Fusion Transformations

Cooperative Sensor Fusion uses multiple events of the same *Event Scheme* to extend the observation of a phenomenon and outputs an event of the same *Event Scheme* with a larger coverage. The extension of coverage typically applies to the time and space domain, but it might also be applied to a scenario specific object domain. Time and spacial cooperative fusion combines multiple sensors observing the same phenomenon. The used operation may either preserve the existing information, such as a concatenation of data or inter- and extrapolation or it combines the information at the cost of increased uncertainty. *Aggregation Transformations* typically use *Cooperative Fusion* internally. Because of some special properties of these *Transformations*, which are discussed in Section 4.5.7. *Cooperative Sensor Fusion Transformations* are *Generic Homogeneous N-Ary Transformations* producing events that provide sensor information in the time space domain, where no sensor information was provided by the existing sensors in the system. The *Interpolation Transformation* uses existing sensor information in form of ASEs in the neighborhood of the required time space point and computes the most likely sensor information at this point. Multiple processing operations are possible to achieve this behavior, ranging from linear and quadratic interpolation by using two and three input events to stochastic estimators such as *Kalman Filters*, described in Section 4.5.8. Depending on the type of interpolation, different amounts of ASE are necessary. In general, these operations increase the uncertainty of the sensor information, as no new information is created, but only estimated based on existing data.

This type of transformation is generic, because it acts on whole events and the specified operations are executed on all attributes contained in the input events. Therefore, the *Event Scheme* of all input events needs to be exactly equal, with the exception of the system model event used by the *Extrapolation Transformation* and the *Kalman Transformation*, which is discussed in Section 4.5.8.

In contrast to the *Complementary Fusion Transformations*, the *Cooperative Sensor Fusion Transformations* need to always be always in the *Transformation DAG* of the resulting *Composite Transformation* for channel. This enables the system to trade certainty of information against delivery of information, because a selection predicate may be unfulfilled by existing events. Interpolation of existing events may provide an appropriate event with an increased uncertainty. As a result, this type of fusion transformations allows the system to use all existing information to fulfill the requirements of the subscriber. Of course subscribers can also express selection predicates to filter out information with a very high uncertainty to limit their input to useful information. The following paragraph explains the four general *Cooperative Sensor Fusion Transformations* of ASEIA: *Concatenation*, *Interpolation* and *Extrapolation*.

Concatenation *Concatenation Transformation* enables the accumulation of multiple homogeneous events to a single event. It is generally used on multi-dimensional attribute values and enables the stitching of camera images or maps. The target of the *Transformation* is to generate data according to the subscribers attribute value type dimension. The

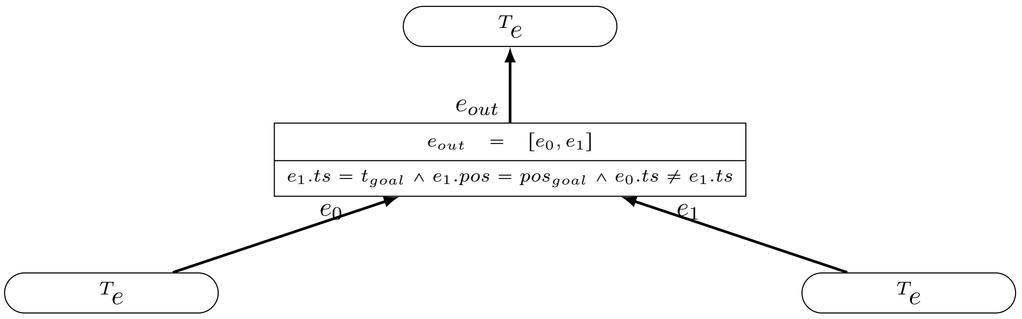

T_e
e_{out}
$e_{out} = [e_0, e_1]$
$e_1.ts = t_{goal} \wedge e_1.pos = pos_{goal} \wedge e_0.ts \neq e_1.ts$
e_0
e_1
T_e
T_e

Figure 4.14.: An example homogeneous transformation concatenating sensor information.

transformation can be subdivided into spacial and timeline concatenation. In case of spacial concatenation, the input matrices of each attribute are combined according to the spacial relation, which needs to be 2-dimensional. Afterwards, the result is clipped to match the size required by the subscriber. The spacial relation is formulated as the interval on the x-y plane that the goal matrix should cover. The input events need to cover this area completely. Therefore, the offset of the sub-matrices of the input events can be computed based on the input events position $e_i.pos$, the target position $e_{out}.pos$, the target matrix dimension $n \times m$ and the input value's dimension $n' \times m'$ to be $(i, j) = \frac{e_i.pos^v - e_{out}.pos^v + e_{out}.pos^u}{e_{out}.pos^u - e_i.pos^u}$. In case the input events matrix exceeds the range, the indices of the matrix will be < 0 or bigger then $\geqslant n$ or $\geqslant m$ and these values will be discarded. In case of temporal concatenation, the input events are ordered according to time and afterwards, the matrix is extended column wise to the right. The resulting transformation is shown in Figure 4.14.

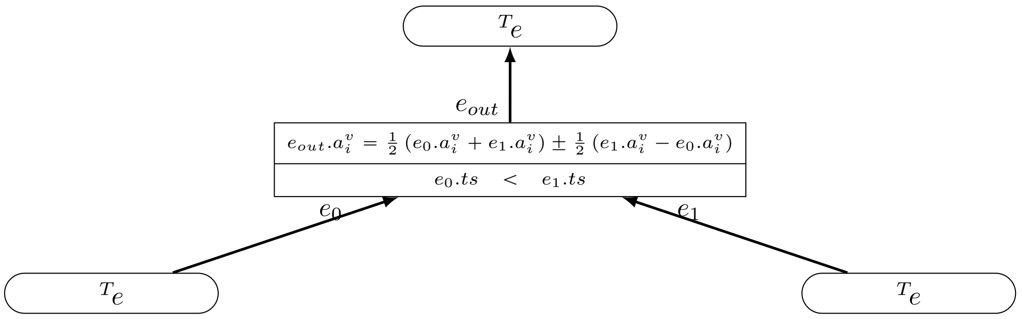

T_e
e_{out}
$e_{out}.a_i^v = \frac{1}{2}(e_0.a_i^v + e_1.a_i^v) \pm \frac{1}{2}(e_1.a_i^v - e_0.a_i^v)$
$e_0.ts < e_1.ts$
e_0
e_1
T_e
T_e

Figure 4.15.: An example homogeneous transformation interpolating sensor information.

Interpolation *Interpolation Transformations* enable the generation of intermediate events by approximating the attributes values of an event at a desired tempo-spatial target *goal* relating to a vectored value of the input events $v = (v_0, \ldots, v_n)$. It is automatically activated whenever n events exist surrounding the target of the filter expression. n depends on the dimensionality d of the filter expression and is expressed as $n = 2^d$. For each dimension i a pair of events $e_{i,0}$ and $e_{i,1}$ is needed that fulfill the following relation $e_{i,0}.v_i < goal_i < e_{i,1}$. To this end, the transformation needs exactly n input events of the same *Event Type* with disjunct time and position and with equal discrete valued attributes. It follows that only the continuous valued attributes values may be different between the input events e_0 and e_1. The intermediate event continues attributes values $a^v_{out,i}$ are generated as the axis-aligned bounding box of the input event's attribute values following Equation 4.50. The one dimensional *Interpolation Transformation* is shown in Figure 4.15.

$$e_{out}.a^v_i = \frac{1}{2}\left(\min_{e \in e_{in}}(e.a^v_i) + \max_{e \in e_{in}}(e.a^v_i)\right) \pm \frac{1}{2}\left(\max_{e \in e_{in}}(e.a^v_i) - \min_{e \in e_{in}}(e.a^v_i)\right) \tag{4.50}$$

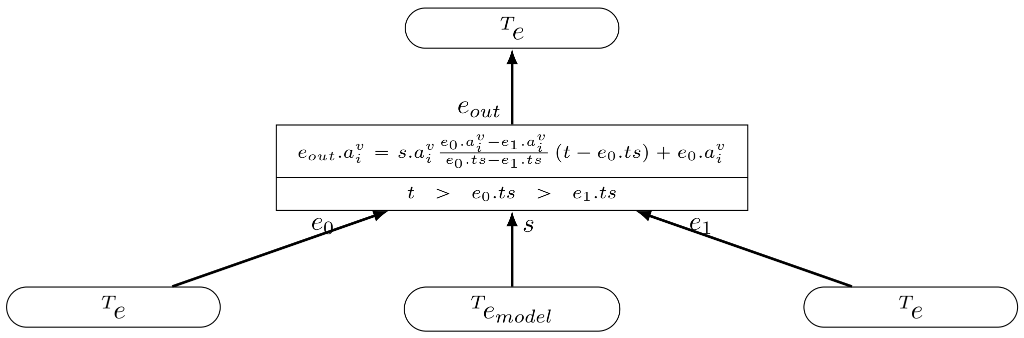

T_e
e_{out}
$e_{out}.a^v_i = s.a^v_i \frac{e_0.a^v_i - e_1.a^v_i}{e_0.ts - e_1.ts}(t - e_0.ts) + e_0.a^v_i$
$t > e_0.ts > e_1.ts$
e_0
s
e_1
T_e
$T_{e_{model}}$
T_e

Figure 4.16.: An example homogeneous transformation extrapolating sensor information.

Extrapolation *Extrapolation Transformations* are similar to *Interpolation Transformations*, but provide sensor information where no sensor information had been available before. Therefore, it is activated if n events $e_{i,0}$ and $e_{i,1}$ exist, which fulfill the following relation $goal_i < e_{i,0}.v_i < e_{i,1}$. Based on each dimensions event pair, a difference quotient is computed estimating the Jacobian Matrix J_j for the attributes values established by the n input events. This matrix contains the linearized estimation of the attributes values change. The target event e_{out} is computed by the multiplication of the Jacobian with the distance vector d. The distance vector contains the distance between the closest event c of the hypercube and the target. The Extrapolation needs an additional input event providing a system model of the physical process to estimate the uncertainty of the resulting output event. This **Model** event contains a matrix for each continuous valued attribute s_j. The resulting event contains attribute values of the form $e_{out}.a^v_j = s_j J_j d + c_j$. The one dimensional *Extrapolation Transformation* is shown in Figure 4.16.

Examples

Distributed Object Tracking : Nowadays cars are equipped with multiple sensors, some of them are cameras, which enable the possibility to locate and identify other cars in the vicinity. Additionally, statically installed sensors such as id detection sensors for toll accounting provide position information for cars on the road, even if these cars are not networked. These information can be used to track a single car by using the temporal *Concatenation Transformation* on the individual **Object** events over the time interval t_{goal} and the object id id_{goal}. The resulting event contains a $n \times m$ matrix which consists of m position vectors of the **Object**.

Position Interpolation : A general problem in event driven systems is timing. As no global notion of periodicity exists, it is very difficult for independently developed components to state time based requirements. An example is a robot subscribing to position information at a certain time $t \pm \epsilon$. Another robot's position publications might occur at $t - \Delta$ and $t + \delta$. Therefore, these publications will not be delivered to the subscriber because they do not fulfill the specified filter expression. However, the information is still in the system. With a loss of certainty it can be concluded, that the publishing robot has moved between the two publications from the first position $e_0.pos$ to the second position $e_1.pos$. As a consequence, both events could be interpolated to fulfill the specified subscription. The interpolation transformation achieves exactly this process in an automatic way and delivers the robot's **Position** as the interpolation of the two events with increases uncertainty.

4.5.6. Concurrent Fusion

Concurrent Fusion Transformations are *Generic Homogeneous N-Ary Transformations*, which may be used to increase the quality of the sensor data by exploiting existing redundancy in the system. To this end, multiple homogeneous input events observing the same phenomenon at the same spatio temporal point need to be present. The redundancy in the events can then be used to minimize the uncertainty of the sensor information if the different events agree on the observation. Without a specific model, improvements can only be achieved for individual values of an multidimensional attribute or for discrete valued attributes. For generic improvement of uncertainty for continuous valued attributes, special filter operations are necessary such as the *Kalman Filter Transformation*, described in Section 4.5.8. The two generic *Concurrent Filter Transformations* are the *Specification Transformation* and the *Majority Vote Transformation*.

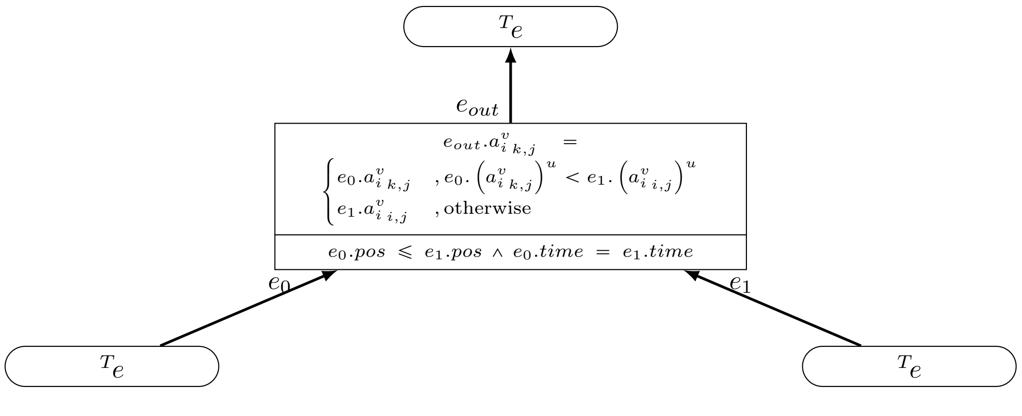

T_e
e_{out}
$e_{out}.a^v_{i\ k,j} =$
$\begin{cases} e_0.a^v_{i\ k,j} & , e_0.\left(a^v_{i\ k,j}\right)^u < e_1.\left(a^v_{i\ i,j}\right)^u \\ e_1.a^v_{i\ i,j} & , \text{otherwise} \end{cases}$
$e_0.pos \leqslant e_1.pos \wedge e_0.time = e_1.time$
e_0
e_1
T_e
T_e

Figure 4.17.: An example homogeneous transformation specifying parts of a multi dimensional sensor information.

Specification The *Specification Transformation* enables the composition of multiple events that agree on the current observation, but with a different uncertainty. It uses the redundancy in observation to minimize uncertainty by selecting for each attributes dimension the value uncertainty complex with minimum uncertainty. This *Transformation* is enabled if the subscriber specifies an uncertainty filter expression on a subscription. The operation is transitive as the elementary uncertainty comparison operation $<$ is transitive. Consequently, a N-Ary *Specification* operation can be broken down into a sequence of dual *Transformations* always using the result of the last *Transformation* as input to the next. The resulting *Generic Dual Homogeneous Transformation* is shown in Figure 4.17. Two events e_0 and e_1 can be used in this transformation if the combined time-space vector of one is completely contained in the other.

Voting Transformation The *Voting Transformations* enable decreasing of the uncertainty of discrete valued attributes. They enable an agreement based on multiple samples formed by the individual decisions or classifications of separate components. The samples need to overlap in time and space. This transformation is selected whenever an uncertainty constraint is stated by the subscriber for a discrete attribute. The consensus uses Bayesian inference,see [93], to compute an uncertainty for each observed value in the sample event set. Depending on the result of this computation, the value with the least uncertainty is selected and output. If two ore more values have an uncertainty 0, at least two input samples with different values have an uncertainty of 0, which is erroneous. In this case they are removed from the sample set and the result will be output. If still no consensus is reached, all equivalently uncertain values are output with maximum uncertainty.

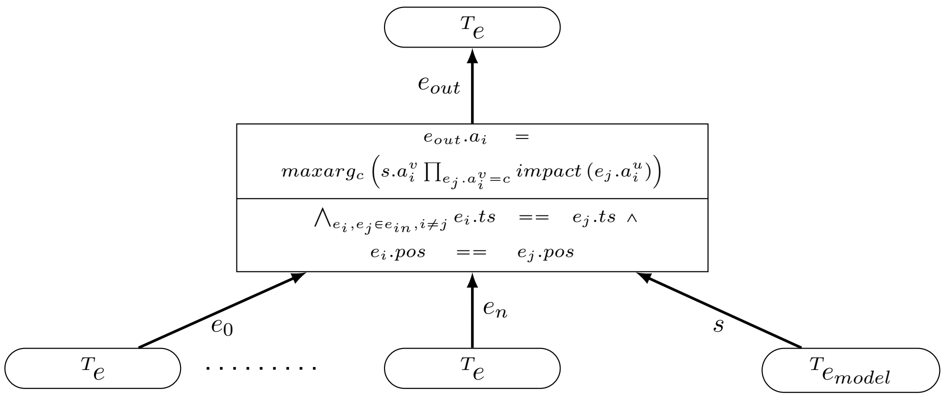
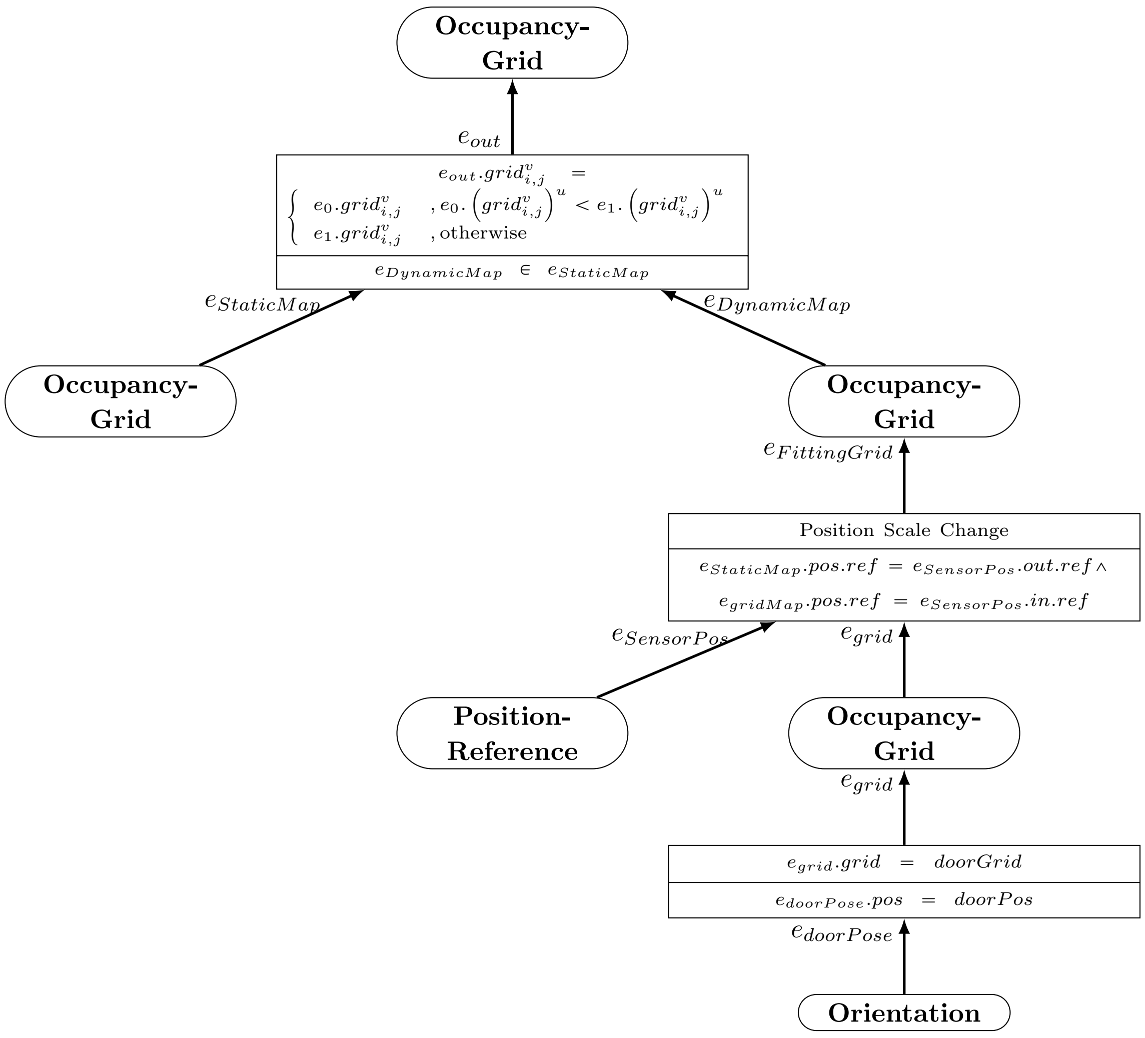

T_e
e_{out}
$e_{out}.a_i \;=\; maxarg_c\left(s.a_i^v \prod_{e_j.a_i^v=c} impact\left(e_j.a_i^u\right)\right)$
$\bigwedge_{e_i,e_j \in e_{in}, i \neq j} e_i.ts \;==\; e_j.ts \;\wedge\; e_i.pos \;==\; e_j.pos$
e_0
e_n
s
T_e
....
T_e
$T_{e_{model}}$

Figure 4.18.: An example homogeneous transformation using multiple events to vote on a discrete attributes a_i value.

Occupancy Grid Overlay : Multidimensional information may contain information of varying quality. This is especially possible for map-like data. If multiple compatible information sets exist, a fusion of them to increase the overall quality of the data is possible. To this end, a combination operation such as the one described in Figure 4.17 may be used. In the robotic scenario a generic tool to represent the map information are occupancy grids, which provide a discretized representation of the ground around a robot as cells, which are either free or occupied. However, sometimes it is not clear whether a cell is occupied which induces an uncertainty in the map. If multiple of these maps exists, which contain heterogeneous information, the overlaying parts of the maps may



Occupancy-Grid
e_{out}
$e_{out}.grid^{v}_{i,j} =$
$\begin{cases} e_0.grid^v_{i,j} & , e_0.\left(grid^v_{i,j}\right)^u < e_1.\left(grid^v_{i,j}\right)^u \\ e_1.grid^v_{i,j} & , \text{otherwise} \end{cases}$
$e_{DynamicMap} \in e_{StaticMap}$
$e_{StaticMap}$
$e_{DynamicMap}$
Occupancy-Grid
Occupancy-Grid
$e_{FittingGrid}$
Position Scale Change
$e_{StaticMap}.pos.ref = e_{SensorPos}.out.ref \wedge$
$e_{gridMap}.pos.ref = e_{SensorPos}.in.ref$
$e_{SensorPos}$
e_{grid}
Position-Reference
Occupancy-Grid
e_{grid}
$e_{grid}.grid = doorGrid$
$e_{doorPose}.pos = doorPos$
$e_{doorPose}$
Orientation

Figure 4.19.: An example homogeneous transformation overlaying multiple occupancy grids to decrease the uncertainty and enhance quality.

be combined by selecting for each cell the value with the smaller uncertainty. This enables the combination of the information from redundant sources and increases the quality of the input data for the navigation algorithm. A static map of a room for example may have very certain values for unmovable objects such as walls, but the certainty of free spaces is low, because mobile objects may occupy them. Occupancy information gathered by sensors in the environment enable the acquisition of more certain information because the generated information is more actual. The *Specification Transformations* allows to combine these two maps according to the contained uncertainty. This enables better navigation decisions of a robot, because the free path can be detected with a higher certainty.

Road State Sensor : In the vehicle scenario it might prove beneficial to detect abnormal road states such as excessive bumps, wetness, ice or gusts. However, a single car observing such phenomenon is very unreliable and will only provide low quality data. However, if multiple cars observe the phenomenon, the observation is much more reliable and of higher quality. If these phenomena are considered discrete road states, the combination of individual information can be done by a *Voting Transformation* of all cars passing through a specific road section formulated as an uncertain position in specific time formulated as an uncertain time.

4.5.7. Aggregation Transformation

Sensor Aggregation Transformation are *Generic Homogeneous N-Ary Transformations*, which combine multiple homogeneous input events to a single output event of same *Event Type* with a largely increased uncertainty, as described in [141]. The large increase in uncertainty is the major difference to *Cooperative Transformations.* As described for *TinyAgg* [104], *Aggregation Transformation* may be separated into three different steps expressed as functions $f_{init} :\rightarrow E_{agg}$, $f_{agg} : E_{agg} \times E \rightarrow E_{agg}$ and $f_{finish} : E_{agg} \rightarrow E$. f_{init} creates an intermediate aggregation event e_{agg} that starts the aggregation process. This function is typically executed by the CPS application if a certain aggregation is wanted. f_{agg} is the *Generic Dual Homogeneous Transformation* combining the intermediate aggregation event with the compatible input events. The aggregation is finished if the stated aggregation range in the intermediate event is filled and the finalization function f_{finish} is executed to produce the final output event. The intermediate event needs to contain a special discrete valued *Attribute* specifying the operation to execute. This is removed by the finalization *Transformation.* Typical *Aggregation Transformations* are *Minimum*, *Maximum*, *Average* and *Count.*

Average The *Average Transformation* is initialized by the creation of a special event. This events needs to contain the goal time and space as interval, a list of attributes it should aggregate and possibly a maximum event count. The aggregation adds each input event's attributes to the intermediate event if the discrete attributes match and the event is contained in the specified time-space range. On finalization of the aggregation, the intermediate event is transformed to the output event by dividing every sum by the count. The resulting *Composite Transformation* is visualized in Figure 4.20.

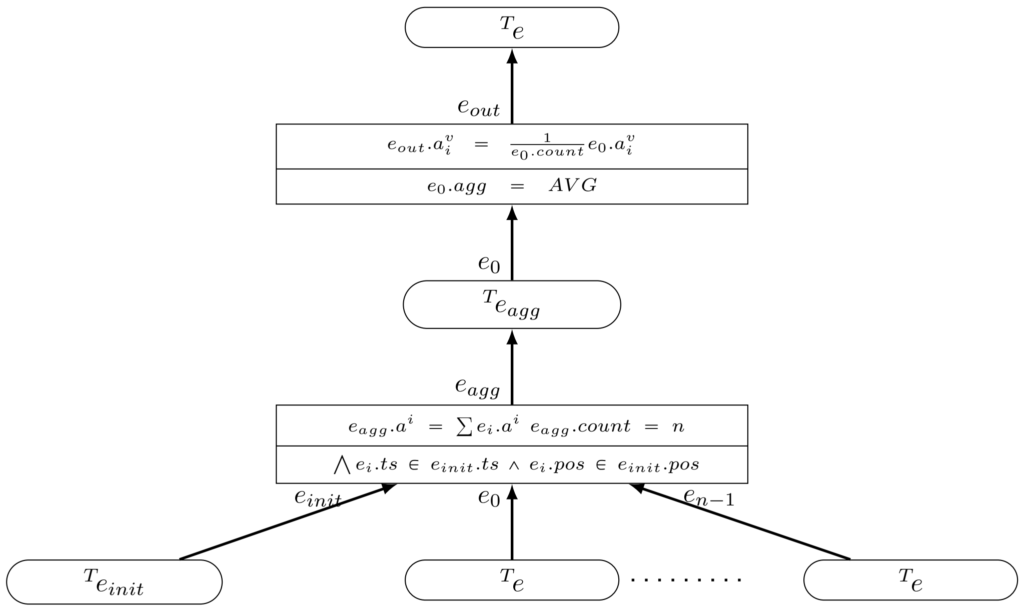

T_e
e_{out}
$e_{out}.a_i^v = \frac{1}{e_0.count} e_0.a_i^v$
$e_0.agg = AVG$
e_0
$T_{e_{agg}}$
e_{agg}
$e_{agg}.a^i = \sum e_i.a^i \; e_{agg}.count = n$
$\bigwedge e_i.ts \in e_{init}.ts \wedge e_i.pos \in e_{init}.pos$
e_{init}
e_0
e_{n-1}
$T_{e_{init}}$
T_e
T_e

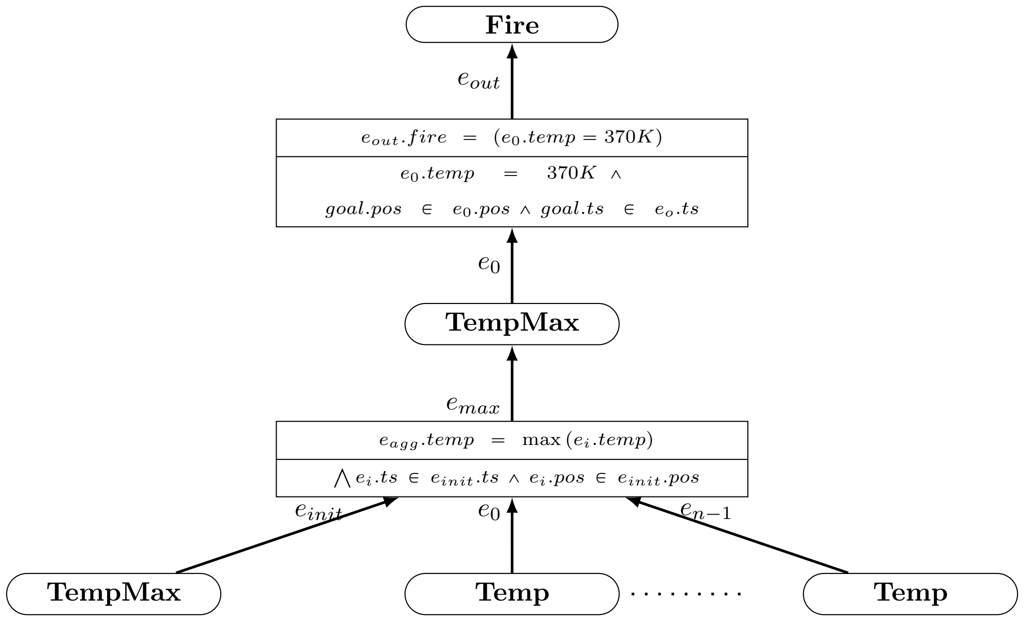
Figure 4.20.: An example homogeneous transformation aggregating multiple values to the average value.

Minimum, Maximum The *Minimum and Maximum Transformations* are *Transformations* computing maximum and minimum of the input events. The CPS need to create an extended event of the input events with an additional discrete value attribute containing either the constant `MIN` or `MAX` and all attributes set to either the maximum value or respectively the minimum value. The input events are included through the execution of element-wise minimum or maximum for each continuous attribute if the discrete attributes match. The finalization function is not necessary in this case, as the intermediate event is already the result event, because of the extension operation used in its creation.

Count and Sum The *Count* and *Sum Aggregation Transformations* are specializations of the *Average Transformation* as the finalization step is ignored. They are initialized by the creation of an initial event with aggregation constant `COUNT` or `SUM`.

Examples

Virtual fire detector of a forest WSN : The forest fire WSN scenario considers the detection of fires in forest. To this end, the WSN needs to acquire, combine and decide on the emerging temperature information created in the network. One possible approach is the aggregation of the temperature over the regions of the forest. This aggregation may use the maximum of the existing temperature to decide on the existence of a fire. Afterwards, a simple complementary fusion transformation may decide on the presence of a fire based on the maximum temperature in the network. To enhance the quality of the result, it may be coupled with a concurrent fusion transformation to decrease



Fire
e_{out}
$e_{out}.fire = (e_0.temp = 370K)$
$e_0.temp = 370K \wedge$
$goal.pos \in e_0.pos \wedge goal.ts \in e_o.ts$
e_0
TempMax
e_{max}
$e_{agg}.temp = \max(e_i.temp)$
$\bigwedge e_i.ts \in e_{init}.ts \wedge e_i.pos \in e_{init}.pos$
e_{init}
e_0
e_{n-1}
TempMax
Temp
Temp

Figure 4.21.: An example aggregation transformation detecting the maximum temperature to detect forest fires.

the uncertainty. The combined heterogeneous n-ary cooperative and complementary transformations usable to implement the virtual sensors is shown in Figure 4.21.

Virtual crowding sensor for robotic navigation : In the robotic scenario it is very useful for robots to detect rooms with lots of dynamic entities. Examples for such rooms are dining halls and elevators. These rooms should be avoided in the navigation, because excessive replanning is necessary as the navigational situations are very complex and dynamic. To detect such rooms, a counting aggregation might be used, which operates on position sensor information. Based on these information the amount of entities in region can be counted and based on the count in relation to the size of the region, a crowding factor may be computed. The transformation used to implement this transformation is a cooperative and complementary n-ary one as shown in Figure 4.22.

Virtual Jam Detect Sensor : Sometimes a single source of information is not enough to decide on the state of a distributed phenomenon. An example are jams on the road. They are characterized by a very low speed of most of the cars in a certain area. An *Average Transformation* on **Speed** events on the road in a certain spatial range will deduce an average speed information from all the cars in the area. Because of the averaging the position of the resulting event grows with each considered event, as averaging using interval arithmetics modifies the center and extends the borders. Afterwards, a simple decision rule may decide if a jam exists in this area or not. This process is shown in Figure 4.23. The resulting transformation represents a homogeneous n-ary cooperative fusion.

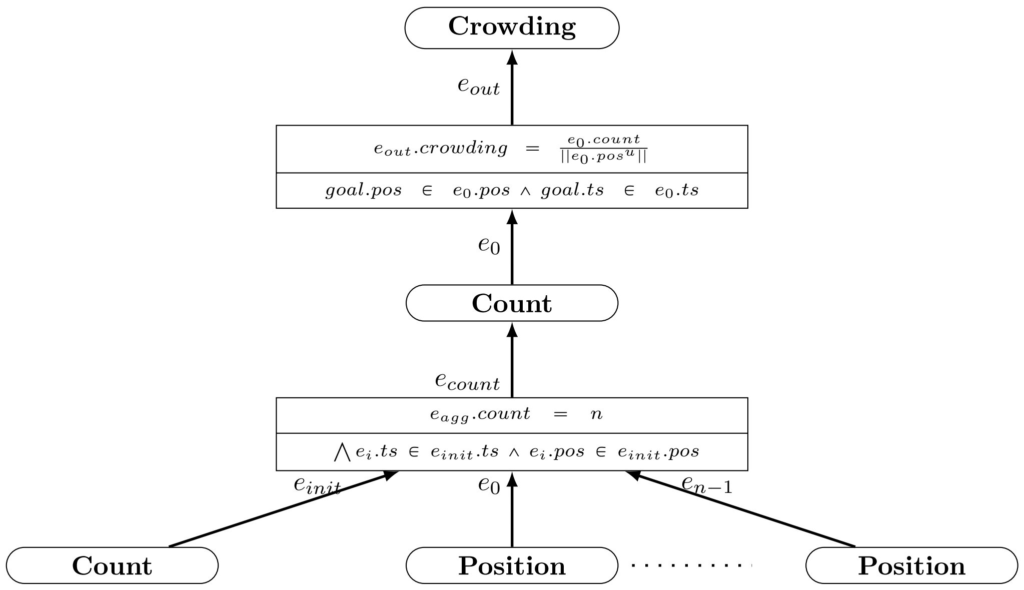

Crowding
e_{out}
$e_{out}.crowding = \frac{e_0.count}{||e_0.pos^u||}$
$goal.pos \in e_0.pos \wedge goal.ts \in e_0.ts$
e_0
Count
e_{count}
$e_{agg}.count = n$
$\bigwedge e_i.ts \in e_{init}.ts \wedge e_i.pos \in e_{init}.pos$
e_{init}
e_0
e_{n-1}
Count
Position
Position

Figure 4.22.: An example aggregation transformation counting the entities in a room to infer a crowding value to be used in navigation.

4.5.8. Kalman Transformations

A *Kalman Filter* is a process that uses observations $\vec{z}_k$ at time k and model information F_k, B_k, Q_k, R_k and H_k on a system to estimate the state of the system $\hat{x}_k$. In its basic form, it models linear systems as a combination of system specific state prediction F_k and user defined input $\vec{u}_k$, see Equation 4.51. To guarantee convergence of the observations and the state predication, all variables of the filter $\vec{z}_k, \vec{u}_k$ and $\hat{x}_k$ are considered to be Gaussian distributed random variables with a covariance. The inputs $\vec{z}_k$ and $\vec{u}_k$ have static covariances R_k and Q_k. The state of the system has a variable covariance depending on the quality of the state prediction and the observations P_k. The remaining matrices B_k and H_k are part of the system model and represent transformations from the input space to the state space and from the state space to the sensor space respectively. The state prediction also updates the covariance of the state variable according to Equation 4.52. The convergence is done by weighted averaging of the updated state with observations based on a dynamic so-called *Kalman Gain* as described in Equation 4.54. Additionally, the resulting covariance is also computed, see Equation 4.55. The dynamic gain is computed as the overlap of the state variable and the observation according to Equation 4.53.

$$\hat{x}_{k|k-1} = F_k \hat{x}_{k-1|k-1} + B_k \vec{u}_k \tag{4.51}$$
$$P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_k \tag{4.52}$$
$$K_k = P_{k|k-1} H_k^T \left(H_k P_{k|k-1} H_k^T + R_k\right)^{-1} \tag{4.53}$$
$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \left(\vec{z}_k - H_k \hat{x}_{k|k-1}\right) \tag{4.54}$$
$$P_{k|k} = P_{k|k-1} - K_k H_k P_{k|k-1} \tag{4.55}$$

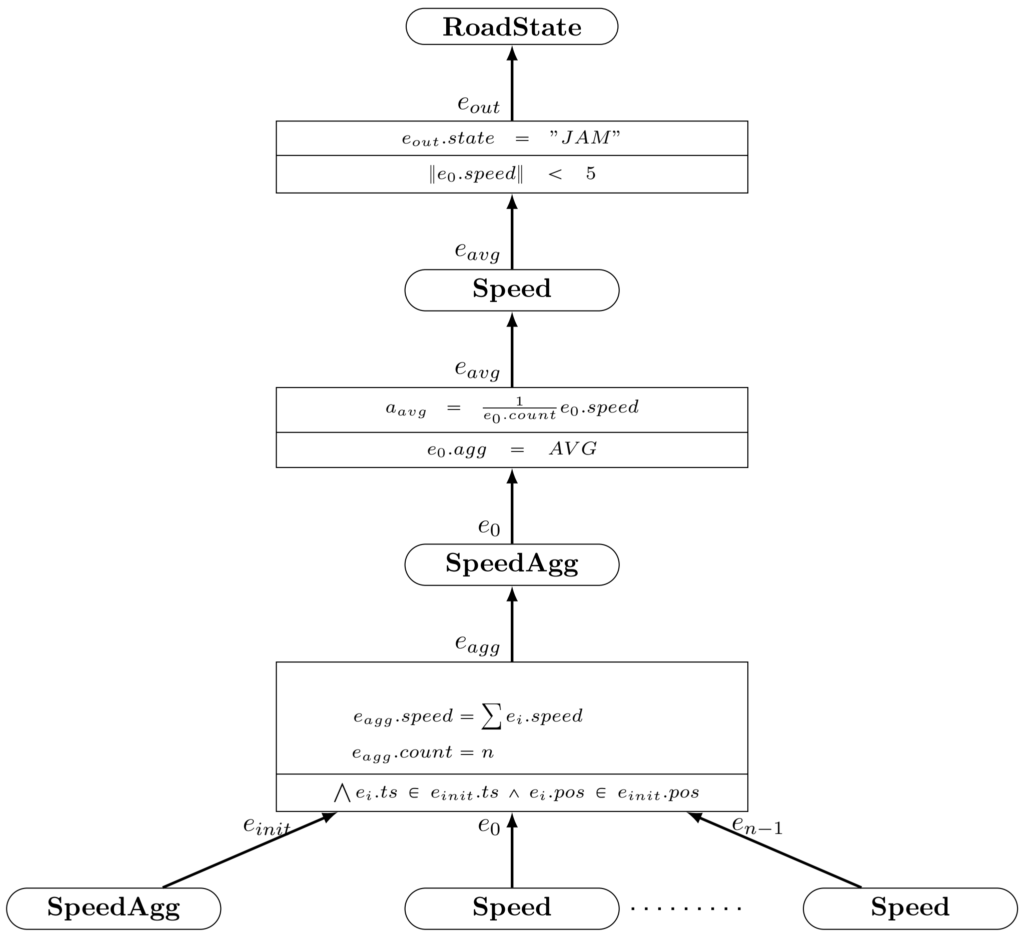
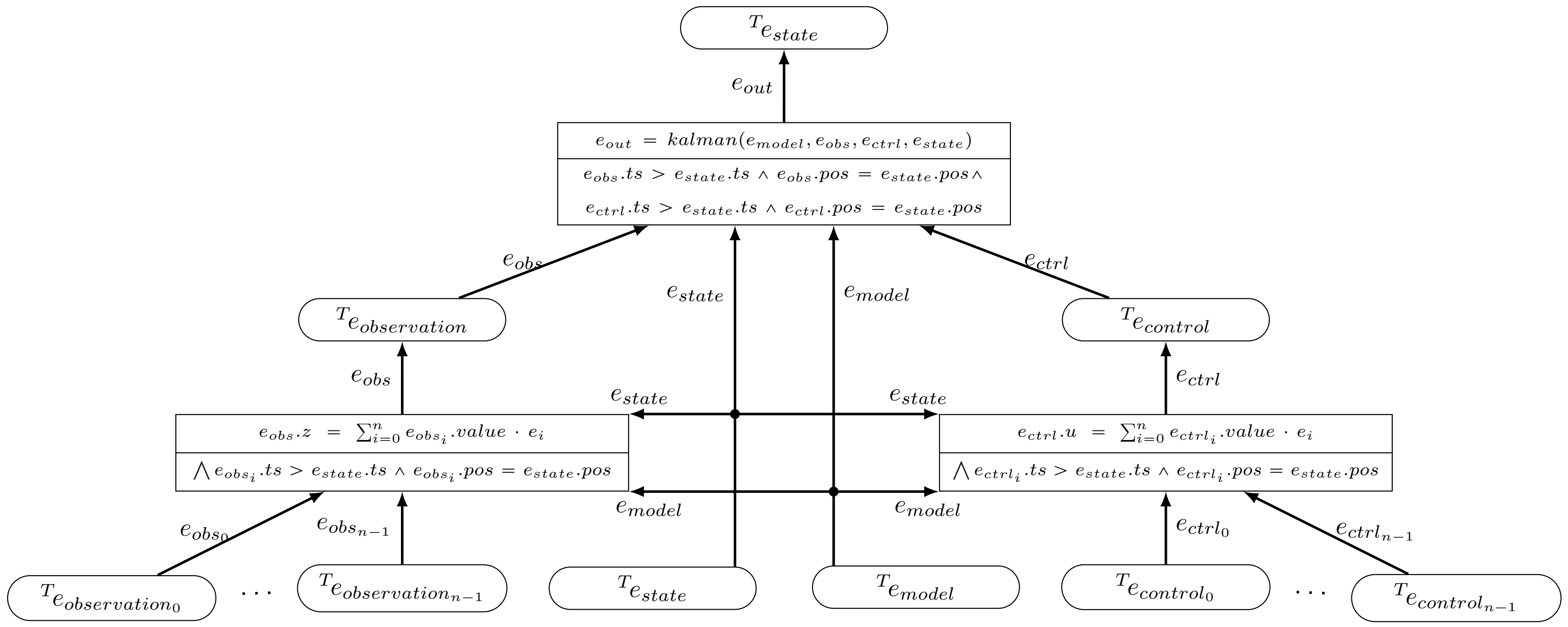


Figure 4.23.: An example homogeneous transformation averaging the speed on a road segment to detect road jams.

To enable a generic *Kalman Filter Transformation*, it is necessary to structure the inputs and outputs in the form of the transformation abstraction. The inputs are the current state of the system, the control input and the observations. Additionally, the system model containing the necessary matrices needs to be supplied as an input event. This event needs to be published only once, as these models are time-invariant. The buffering of the transformation keeps it, so that it may be used in every execution of the transformation. The output of the Kalman is the state of the system with an updated time stamp, which is also used to prevent infinite recursion through an appropriate filter expression. The generic Kalman Transformation is shown in Figure 4.24.



$T_{e_{state}}$
e_{out}
$e_{out} = kalman(e_{model}, e_{obs}, e_{ctrl}, e_{state})$
$e_{obs}.ts > e_{state}.ts \wedge e_{obs}.pos = e_{state}.pos \wedge$
$e_{ctrl}.ts > e_{state}.ts \wedge e_{ctrl}.pos = e_{state}.pos$
e_{obs}
e_{state}
e_{model}
e_{ctrl}
$T_{e_{observation}}$
$T_{e_{control}}$
e_{obs}
e_{ctrl}
e_{state}
e_{state}
$e_{obs}.z = \sum_{i=0}^{n} e_{obs_i}.value \cdot e_i$
$\bigwedge e_{obs_i}.ts > e_{state}.ts \wedge e_{obs_i}.pos = e_{state}.pos$
$e_{ctrl}.u = \sum_{i=0}^{n} e_{ctrl_i}.value \cdot e_i$
$\bigwedge e_{ctrl_i}.ts > e_{state}.ts \wedge e_{ctrl_i}.pos = e_{state}.pos$
e_{model}
e_{model}
e_{obs_0}
$e_{obs_{n-1}}$
e_{ctrl_0}
$e_{ctrl_{n-1}}$
$T_{e_{observation_0}}$
...
$T_{e_{observation_{n-1}}}$
$T_{e_{state}}$
$T_{e_{model}}$
$T_{e_{control_0}}$
...
$T_{e_{control_{n-1}}}$

Figure 4.24.: Figure showing transformation graph of a generic linear Kalman Filter transformation.

A typical problem of distributed event based systems is the unsynchronized timestamps of these inputs. Therefore, it is useful to execute the Kalman Filter whenever new information arrives. However, this needs some adoptions to the Kalman Filter's structure, as observations and control input may not be fed as a single vector, but as multiple individual values. On the assumption that the observations and the control inputs are uncoupled, the vectors can easily be separated in a linear combination of the observation/control input and the Cartesian unit vectors according to Equation 4.56.

$$\vec{z}_k = \begin{pmatrix} z_{0_k} \\ \vdots \\ z_{n-1_k} \end{pmatrix} = \sum_{i=0}^{n} z_{i_k} e_i \tag{4.56}$$

Because the Kalman Filter is used in an event driven system, it will not necessarily be executed with constant Δt. The solution is the separation of the system matrices F_k, B_k into a time step aware part F'_k, B'_k and a matrix $0 \in 1^{m \times n}$ defining the application of the Δ_t and an exponent matrix $E \in Q^{m \times n}$ applying the Δt_k with the correct exponent to all components of F'_k or B'_k. The same replacement needs to be done for the covariance matrix Q_k.

$$F_k = (\Delta t_k O_F)^{E_F} F'_k \tag{4.57}$$
$$B_k = (\Delta t_k O_B)^{E_B} B'_k \tag{4.58}$$
$$Q_k = (\Delta t_k O_Q)^{E_Q} Q'_k \tag{4.59}$$

The final necessary adaptation necessary is the transformation of the ASEs uncertainty to Gaussian distribution's standard deviations. This is can be achieved through the specification of an observation confidence probability α. This allows the computation of the standard deviation based on the uncertainty u_{i_k} of the observation, according to Equation 4.60. The resulting variances of the observation input can be used to formulate the measurement covariance matrix to $R'_k = R'\vec{\sigma_k}$

$$\sigma_{i_k}(u) = \frac{1}{z(1 - {}^{\alpha}\!/_{2})} u_{i_k} \tag{4.60}$$

By combining everything one achieves the formulation of the linear varying time-step separated discrete Kalman Filter transformation in Equation 4.63.

$$\hat{x}_{k|k-1} = (\Delta t_k O_F)^{E_F} F'_k \hat{x}_{k-1|k-1} + (\Delta t_k O_B)^{E_B} B'_k \vec{u}_k \tag{4.61}$$
$$P_{k|k-1} = \left((\Delta t_k O_F)^{E_F} F'_k\right) P_{k-1|k-1} \left((\Delta t_k O_F)^{E_F} F_k\right)^T + (\Delta t_k O_Q)^{E_Q} Q'_k \tag{4.62}$$
$$\tag{4.63}$$

As a result, the configuration space of the generic *Kalman Transformation* consists of the E_F, E_B, E_Q matrices defining the impact of the time step on the state F'_k, control

B_k and covariance Q_k matrix. Additonally, the sensor transformation matrix H_k and the confidence probability of the sensor input α need to be specified. The state space of the kalman is defined through a vector X of *AttributeID*s, whereas the individual sensor inputs are configured as a vector s of *AttributeID*s to define the sensor input of the *Kalman Filter Transformation*.

Example: Distributed ACC-Sensor Cars in the vehicle scenario primarily need to trust their integrated sensors, as these are available without latency and with lower uncertainty. However, in some situations such as mountain roads with limited front visibility some sensors are unable to detect other cars because of occlusion. In these scenarios it might be beneficial to use other cars' sensors as additional data sources. A *Kalman Filter Transformation* may be used to combine the existing data in a useful manner. In this case the incoming event information is assessed based on the quality and the age using the system specific **Model** event. The local information and the remote information are then averaged according to their uncertainty. The described linear Kalman may be used if a road coordinate system is used. Additionally, this eases the comparison of the compatibility of the incoming events. This *Transformation* enables a car to dynamically combine the best incoming data of its surroundings to provide its internal CPS driving application with better information. The resulting *Kalman Transformation* is shown in Figure 4.25.

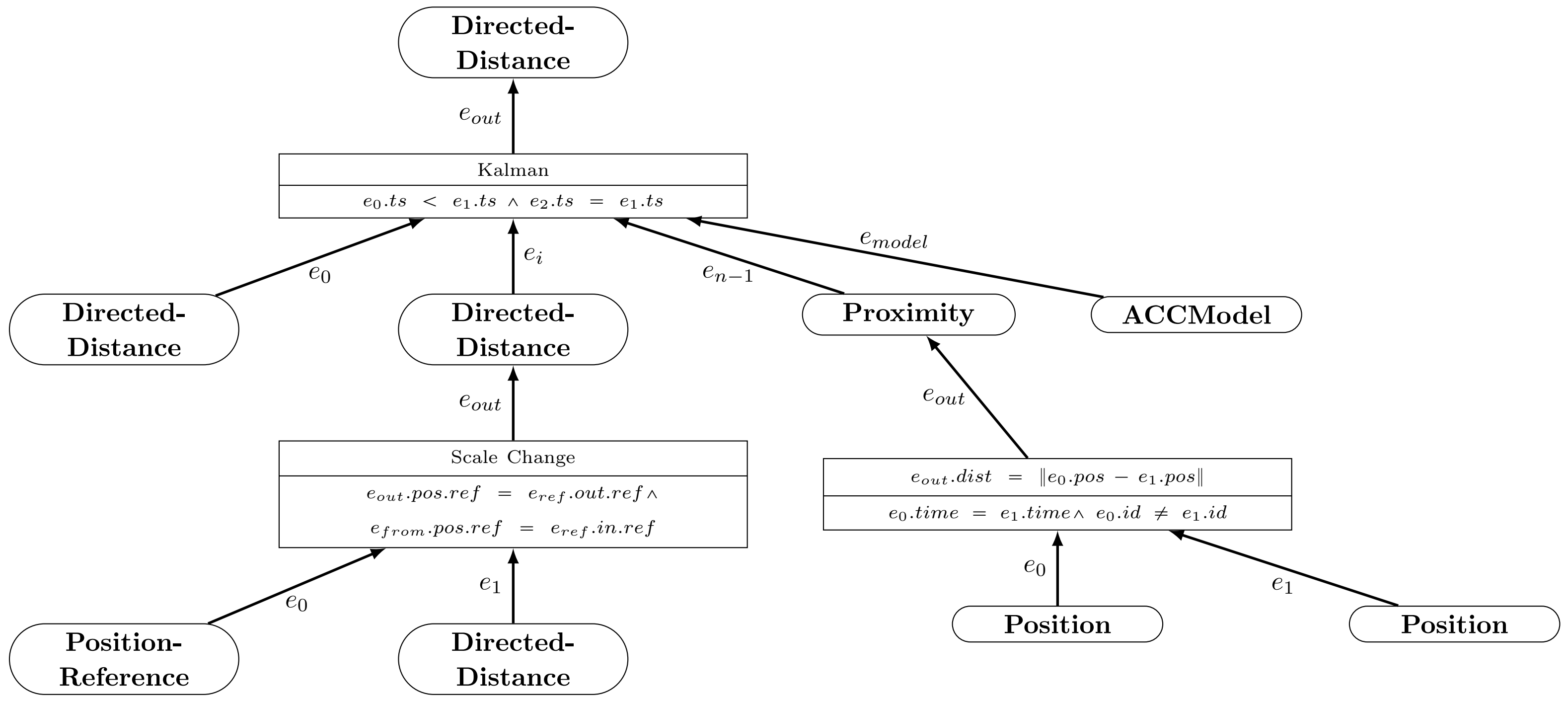


Figure 4.25.: An example homogeneous transformation using local, remote and inferred **DirectedDistance** information to enhance the ACC behavior of an autonomous car.

4.5.9. Uncertainty-aware Hybrid Clock Synchronization

As described in Section 2.1.3, time is one of the most important context attributes for CPS. The distribution of the processing in a general CPS results in multiple independent clocks existing in the system. These clocks are unsynchronized and therefore, the reference and/or scale of their created time information is incompatible. To enable a processing of the information created in the different nodes these heterogeneous time information need to be translated to a homogeneous scale. However, the distribution of time information in the system is difficult to achieve without the addition of uncertainty created from propagation and processing in the different nodes. The following section will investigate an uncertainty-aware hybrid clock synchronization system based on the described transformation systems for CPS. The major goal of this mechanism is the creation of a time transformation outputting additional uncertainty information. To provide references of the used ideas and comparison the most important clock synchronization algorithms are briefly described in the next paragraphs.

Reference Broadcast Synchronization (RBS) Reference Broadcast Synchronization by Elson et al. [62] is an internal synchronization mechanism exploiting physical broadcasts. The synchronization averages the current clock values of the participating nodes. To this end, an initiating node transmits a NOW-message to all participating nodes. This message is the indication for the other nodes to aquire a local time stamp from their clocks, which are exchanged afterwards. The exchange of the time stamps is not critical towards the synchronization's performance , because the critical path is reduced to the transmission time of the NOW-message and the local processing time on each node until the local time stamp is taken. This provides very tight synchronization in single-hop scenarios as long as the computation time is bounded. However, in worst case $\mathcal{O}(n^2)$ messages are needed to exchange the time stamps between all nodes.

In summary, *RBS* provides tight synchronization bounds for single-hop environments. However, it reacts very sensitively to mobile or faulty nodes. This is caused by the used averaging mechanism of the protocol. The contribution of all nodes to the averaged new time and clock rate may create large shifts of agreed global clock whenever one node's clock is far off. This is especially problematic whenever a node is only a temporary member of the broadcast group.

CesiumSpray (CS) CesiumSpray by Verissimo et al. [166] is a pseudo-hierarchical non-averaging hybrid clock synchronization. Additionally, it provides strong failure resilience in real-time networks. The baseline idea of this system is the synchronization of groups of nodes towards an external reference. Since groups of nodes and external references are used, the approach needs and internal an an external synchronization mechanism. The authors proposed GPS receivers as the external reference, but other time sources such as DFC77 receivers are possible. The external reference forms a backbone of synchronization networks, where the group members are leaf nodes.

CesiumSpray is resilient and real-time capable, because the failure of a group member has no influence on other nodes and the failure of an external reference node only influences the local group, unless no other external receiver node exists. It provides excellent failure resilience with a proven upper bound on the synchronizations precision and accuracy. The precision is dependent on the tightness of the network and the real-time capabilities of the used platform. The drawbacks of the approach consist in the need for real-time capable networks and operating systems to use the strong failure resilience and the need for an external time source providing the "multi-hop" capability.

Delay Measurement Time Synchronization Protocol (DMTS) The Delay Measurement Time Synchronization Protocol described by Ping [124] is a modified version of RB, which exploits low-level hardware access and uses a master-slave architecture. It extends the NOW-message with a time stamp taken and inserted just before sending. The master of the synchronization sends this message to all slaves, which can directly use the time stamp to update their time. Therefore, the exchange of the individual local time stamps is omitted and the message count is heavily reduced.

DMTS provides a high precision clock synchronization in single-hop neighborhoods as well as multi-hop synchronization with slowly degrading performance. It solves the large amount of message necessary for a single synchronization of *RBS*. However, in-depth knowledge of the needed hardware and communication mechanisms as well as low-level hardware access is needed to implement it, which makes deployment in arbitrary systems difficult. The robustness is limited as only a single time stamp is communicated from master to its neighborhood, which leaves the protocol open to omission failures and faulty nodes spreading wrong clock values when elected.

Continuous Clock Synchronization in Wireless Real-time Applications (CCS) The Continuous Clock Synchronization for Wireless Real-time Applications by Mock et al. [112] is a non-averaging master-slave synchronization method extending the basic clock synchronization of the 802.11 standard [10]. This is itself the adoption of the approach of Gergeleit and Streich [72] towards 802.11 networks.

The major change is the slow adaptation of the offset between master and slave clocks to avoid jumps in the time domain of the different nodes. Additionally, the beacon messages used for the synchronization are embedded into the 802.11 access point protocols, which minimizes overhead.

This approach enables continuous clock synchronization without gaps in the time base suitable for real-time applications. Additionally, it provides better precision than the baseline 802.11 synchronization mechanism without additional message overhead together with failure resilience and a guaranteed precision for a known omission degree. However, it is generally only useful for single-hop environments with a dedicated access point.

Probabilistic Clock Synchronization Service (PCS) The Probabilistic Clock Synchronization Service by PalChaudhuri et al. [119] is an extension of *RBS* enabling a dynamic trade-off between synchronization precision and message overhead. This approach transmits n NOW-messages in one synchronization round, which are used to derive the skew of the sender's and the receiver's clock through linear regression. The results are combined and transmitted back to the receivers in range. The results are compared by each node with their own data, which enables the agreement on the local time. The number of needed NOW-messages can be derived based on a normal distributed synchronization error. This normal distribution is modelled with zero mean and a standard deviation of σ. Based on this distribution, the authors analytically derive the probability $P(|\epsilon| < \epsilon_{max})$ of the synchronization error to be lower than a specified value ϵ_{max}. This relation can be exploited to evaluate the number of messages needed for a synchronization round.

The approach provides a dynamic trade-off between message overhead and synchronization precision even in multi-hop scenarios. The biggest issue with the approach is the acquisition of the normal distribution parameters of the synchronization errors if the error is Gaussian. Additionally, it is difficult to handle dynamic synchronization errors as imposed by changing environments with the system.

Time Synchronization in Ad-Hoc Networks (TSAN) Römer's Time Synchronization in Ad-Hoc Networks [135] is based on Christian's Algorithm [53]. It is a non-averaging internal synchronization using pair-wise offset estimation. It estimates the round trip time of a messages between sender and receiver and ultimately tries to order events created by the system. In contrast to Christian's Algorithm which proposed a dedicated server for clients to communicate to, this approach attaches time stamps to events communicated in the network, which induces zero message overhead. However, not all events are acknowledged by the receiver, which might create large durations between events flowing in both directions between two nodes. This is mitigated by the insertion of additional dummy events in case the duration grows too large, which creates additional overhead.

TSAN supports multi-hop synchronization directly, since it measures round-trip time from sender to receiver and back. The basic concept makes no assumption on the amount of hops between sender and receiver and works with an arbitrary amount of hops.

TSAN applies a very loose multi-hop synchronization with an ideal message overhead of 0 in a multi-hop network. Unfortunately, the real message overhead is heavily dependent on the actual communication in the network and is therefore very hard to estimate for a real system. The used time intervals together with the *MAYBE*-results of the event ordering provide the protocol with robustness against large synchronization errors caused by message loss or unpredictable delays.

Protocol Summary The individual problems and features of the protocols are summarized in Table 4.1, which shows that none of the described approaches fully solve the problem of multi-hop uncertainty aware clock synchronization. However, each protocol provides unique solutions to certain sub-problems, which are used in the following Uncertainty-aware Hybrid Clock Synchronization (UHCS).

WSN provide the hardest environment for clock synchronization mechanism, as they provide sparse connectivity in time and space between nodes together with a limited life-time of nodes. Therefore, the *UHCS Transformation* is defined based on their requirements and limitations. On the one hand, the synchronization needs to be scalable, while on the other hand, the overhead may not exceed a certain threshold to safe battery and prevent an overload of the network. The following *UHCS Transformation* is a *Scale Change Transformation* executed in every broker of the system to enable the time stamps to be converted to appropriate domains. The *Scale Change Transformation* needs a reference source, which provides the necessary conversion information on the offsets and skews between the clocks. To this end, an estimation mechanism needs to be established which provides the necessary **Reference**. Most of the approaches discussed in previous paragraphs favor either synchro-

Protocol	Precision	Multi Hop Capability	Message Overhead	Robustness
CS	medium	inherent	$\mathcal{O}(n)$	medium
RBS	high	none	$\mathcal{O}(n^2)$	fragile
DMTS	high	possible	$\mathcal{O}(n)$	fragile
CCS	high	possible	$\mathcal{O}(1)$	robust
PCS	medium	inherent	dynamic	medium
TSAN	low	inherent	0 - $\mathcal{O}(1)$	medium

Table 4.1.: Comparison of the discussed clock synchronization protocols.

nization accuracy (and with it certainty of offset and skew estimation) over minimization of communication overhead. This generic trade-off can only be overcome if specific WSN base topologies are exploited, which enable better solutions. One interesting topology is the cluster tree structure of IEEE 802.15.4 networks [12] in beacon-enabled mode. This mode divides the nodes in groups called Personal Area Networks (PANs), which have an individual coordinating instance managing the internal communication. The individual PANs communicate only through their respective coordinators, as visible in Figure 4.26. This hierarchical network structure may also be found in other types of networks like Bluetooth scatter nets as proposed by the Bluetooth standard [38]. Interestingly, P/S networks using an overlay structure also show this pattern. In this case the PAN coordinators are typically the brokers of the network. The remainder for the section considers such an hierarchical structured network with a coordinator and its directly connected local neighbors. In the following, the coordinator will be called *Master* and the topologically adjacent neighbors *Slaves*. Each cell of a single *Master* and its *Slaves* is called a *Cluster*.

The time model, described in Section 4.3.3, assumes that the synchronization may have a decreasing precision based on topological distance between nodes in the network. Consequently, a hybrid clock-synchronization is proposed [156], which consists of a tight synchronization mechanism for each Master called *Intra Cluster Synchronization* and a loose synchronization mechanism between the Masters, called *Inter Cluster Synchronization*.

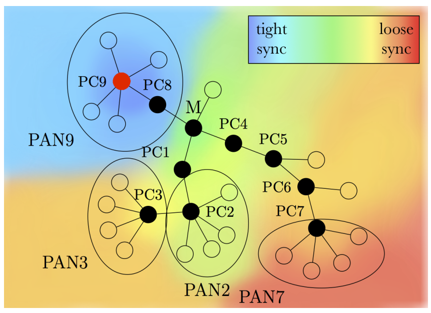


Figure 4.26.: Example cluster tree structure of an IEEE 802.15.4 Network. The colors indicate the topological distance between each node and $PC9$.

Intra Cluster Synchronization The *Intra Cluster Synchronization* is executed below the actual *ASEIA Transformation Engine* and directly synchronizes the clock of the *Slaves* in the *Cluster* to the clock of the *Master*. No *Transformation* is necessary in this case as all time stamps are created in the same *Scale* with the same *Reference* and a known *Continuous Uncertainty*.

The Intra Cluster Synchronization is based on the *CCS* approach, see Section 4.5.9. Therefore, each Slave $P_{s_j} \in Slaves$ has a virtual synchronized clock $VC_{s_j}(t)$. This clock uses the time stamps created by the node's hardware clock $C_{s_j}(t)$ and modifies it based on the current rate $r_{s_j,i}$:

$$VC_{s_j}(t) = r_{s_j,i}\left(C(t)_{s_j} - C(t_i)_{s_j}\right) + VC_{s_j}(t_i) \tag{4.64}$$

The task of the Intra Cluster Synchronization is the estimation of the parameter $r_{s_j,i}$ for each slave at each synchronization round i. To this end, a periodic exchange of information on the current clock values of the *Slaves* is necessary. The implementation of this exchange depends on the used communication system. The 802.15.4 standard, for example, allows a PAN Coordinator to attach additional information to the beacon frame. This may be used to attach a 64bit time stamp $t_{c,i}$ to each beacon b_{i+1} transmitted by the coordinator. The attached time stamp represents the *Masters*'s time of successful transmission of the last beacon. This time stamp together with the local reception time of the last beacon $t_{s_j,i}$ is then evaluated by each *Slave* P_{s_j} to compute a new rate $r_{s_j,i}$.

As described by *DMTS*, see Section 4.5.9, hardware knowledge may be used to provide the needed local time stamps with the necessary accuracy. The accuracy of this time stamp directly influences the achievable synchronization of the *Slaves* to the *Master*. The 802.15.4 standard provides the `PD-Data.confirm` primitive as a local event indicating completion of a transmission. The time of this event can be used as the source of the time stamp $t_{c,i}$. On reception of the beacon each *Slave* P_{s_j} takes a local time stamp $t_{s_j,i+1}$. The networks tightness τ together with the internal computation time t_{comp} of the nodes limits the accuracy of the local time stamps. This computation time is mitigated by the `PD-DATA.indication` primitive of the 802.15.4 standard. As most low level communication systems provide similar mechanism, an appropriate optimization should be possible for each CPS. Therefore, the UHCS considers t_{comp} to be very small. Consequently, the time difference between creation of the local time stamps is bounded by τ.

After acquiring the time stamp for the actual synchronization round the *Slaves* compute the offset $\delta_{j,i} = t_{c,i} - VC(t_{s_j,i})$ between their previous local time stamp $VC(t_{s_j,i})$ and the time stamp transmitted by the *Master* $t_{c,i}$. This is used to compute a new rate $r_{s_j,i+1} = 1 + k_r\delta_{j,i}$ for the node's virtual clock to compensate the offset, with k_r being a proportional factor controlling the rate of adaption.

The Intra Cluster Synchronization provides continuous clock synchronization between the *Master* and its *Slaves* within the *Cluster*. The overhead should be minimal since depending on the used communication system embedded of the time stamps into existing periodic communication can be exploited by following *TSAN*s method.

Inter Cluster Synchronization Diverging from the Intra Cluster Synchronization, as described in the previous paragraph, a *Master* never modifies its own clock. Instead, every event received by a *Master* P_{c_r}, which is transmitted by another adjacent *Master* P_{c_s} is transformed to the time domain of the *Master's* clock $C_r(t)$, as proposed by *TSAN* and according to the Time Model, see Section 4.3.3. To achieve this, the *Master* P_{c_r} needs to estimate its *Reference Offset* $\Delta r_{r,s_i}$ against the *Time Attribute Scale* of its neighboring *Master's* Virtual Clocks $VC_{r,s_i}(t)$. ASEIA assumes the *Time Attribute Scale* to be linear scaled with the parameters d and n representing the scaling quotient $f = \frac{n}{d}x + r$. According to the general definition of the *Scale Change Transformation*, see Section 4.5.2, a transformation requires the scaling function f_s and the inverse scaling function f_r^{-1} to transform the events' *Time Attributes*. The parameters d_s, n_s, d_r and n_r are part of the *Event Type* information and exchanged on channel creation. Consequently, only the reference delta $\Delta r_{s,r}$ need to be estimated dynamically.

The virtual clocks are handled similarly to the Intra Cluster Synchronization, since all beacons of all adjacent *Master*s P_{c_s} are received by *Master* P_{c_r}. On reception of a beacon containing a time stamp $t_{s,i}$, P_{c_r} acquires a local time stamp $t_{r,s,i+1}$. This enables the computation of the offset $\delta_{r,s,i} = t_{s,i} - t_{r,s,i}$ between P_{c_s} and P_{c_r}. Afterwards, P_{c_r} updates the rate $r_{r,s,i} = 1 + k_r\delta_{r,s,i}$ for the virtual clock $VC_{r,s}(t)$ towards P_{c_s}. Therefore, each *Master* has an internal list of virtual clocks following the clocks of each adjacent *Master* as visible in Figure 4.27. The generated *Time Scale Reference Events* enable the generic *Scale Change Transformation* to be executed in each *Master* on reception of an event originating in another *Cluster*.

In case of multi-hop communication the event's time stamp is always transformed to the receiver's clock domain before forwarding it. Consequently, all nodes only need to estimate the offset by using a virtual clock for their directly adjacent neighbors. An example scenario is shown in Figure 4.27. In this picture three *Master*s are in direct vicinity and estimate their offset by using virtual clocks. An event is transmitted from PC_0 to PC_2 via PC_1. During the forwarding of the event the time stamp of the event is adjusted by the estimated offsets to transform it to the local time domain of the current node.

Performance Estimation

The performance of the synchronization depends on certain network and node parameters such as the tightness of the network (including propagation speed and message delivery time) τ, the drift of the nodes ρ and the algorithm parameter k_r. To analyse the behavior of the algorithm, two clocks are defined: one for the transmitter of the beacon $C_s(t) = C_s(t_{s,i}) + (t - t_{s,i})(1 \pm \rho)$ and one for the receiver of the beacon $C_r(t) = C_r(t_{r,i}) + (t - t_{r,i})(1 \pm \rho)$. The offset between these two nodes is described as the difference between local clock of the sender and the Virtual Clock of the receiver as $o_{r,s}(t) = C_s(t) - VC_r(t)$. The receiver cannot observe this offset, but based on the algorithm it uses the visible offset $\delta_{r,s}(t) = o_{r,s}(t) \pm \tau$ based on the time stamp contained in the beacon. Consequently, the virtual clock's propagation formula may be rewritten to:

$$VC_r(t) = VC_r(t_{r,i}) + (C_r(t) - C_r(t_{r,i})(1 + k_r\delta_{r,s}(t_{r,i})) \tag{4.65}$$

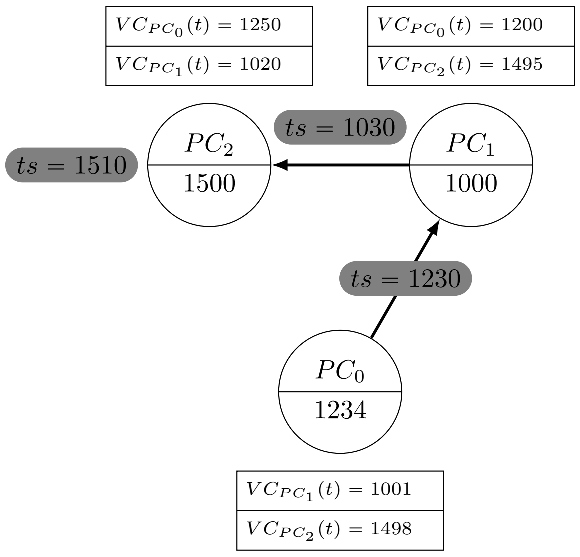


Figure 4.27.: An example network composed of three *Master*s with their respective virtual clocks estimating their offset towards each other. An event transmitted from PC_0 to PC_2 is shown in grey including its time stamp.

Entering $VC_r(t)$ in $o(t)$ enables replacing of $C_s(t) - VC_r(t_{r,i})$ with $o_{r,s}(t_{r,i}) + (t - t_{r,i})(1 \pm \rho)$. Together with the simplification of $C_r(t) - C_r(t_{r,i})$ to $(t - t_{r,i})(1 \pm \rho)$ one obtains:

$$o_{r,s}(t) = o_{r,s}(t_{r,i}) - (t - t_{r,i})(1 \pm \rho)k_r\delta_{r,s}(t_{r,i}) \pm (t - t_{r,i})2\rho \tag{4.66}$$

By substitution of $\delta_{r,s}(t_{r,i})$ and $(t - t_{r,i})$ to Δt as well as factoring out $o_{r,s}(t_{r,i})$ the following equation is achieved:

$$o_{r,s}(t) = o_{r,s}(t_{r,i})(1 - \Delta t k_r) \pm (\Delta t(k_r \rho o(t_{r,i}) + \rho(2 + \tau) + k_r\tau)) \tag{4.67}$$

To enable a normal flow of time, the rate of the Virtual Clock needs to be bigger than zero. Therefore, the value of k_r must not exceed $\frac{1}{max(\delta_{r,s}(t))}$. The propagation of the offset may now be separated into an offset compensating part: $o^-_{r,s}(t) = o_{r,s}(t_{r,i})(1 - \Delta t k_r)$ and an offset increasing part: $o^+_{r,s}(t) = \pm\Delta t(k_r \rho o_{r,s}(t_{r,i}) + \rho(2 + \tau) + k_r\tau)$. The offset decreasing part will converge towards zero, because it resembles the geometric sequence $a^{i+1} = a^i q, 0 < q < 1$. The offset inducing part can be separated into an offset dependent and offset independent part. The value of the offset dependent part $\Delta t k_r \rho o_{r,s}(t_{r,i})$ depends very much on the offset between sender and receiver at the last synchronization and the time passed since this synchronization. For tightly synchronized nodes the impact of this part will tend towards zero.

The guaranteed precision of the method for tightly synchronized nodes is the absolute value of the offset independent part of $o_{r,s}(t)$. The resulting precision of the Intra Cluster Synchronization is $\pi_{intra}(t) = \Delta t(\rho(2 + \tau) + k_r\tau)$.

The offset independent part is $\Delta t(\rho(2+\tau)+k_r\tau)$. In case no beacon loss occurs Δt will be approximately twice the beacon Δt_b interval, because every beacon contains the sender's time stamp of the last round and the maximum offset is always reached directly before the next beacon arrives. The resulting precision in case of OD beacon omissions will therefore be:

$$\pi_{intra} \leqslant (2+\mathrm{OD})\Delta t_b(\rho(2+\tau)+k_r\tau) \tag{4.68}$$

The value k_r can be viewed as a trade-off factor choosing between fast synchronization and robust synchronization. This is caused by the presence of the factor in the offset compensating part, where it decreases the existing offset stronger when it is bigger. On the other hand, bigger k_r values will increase the results of the offset increasing part depending on the values of ρ and τ.

For the offset estimation of the adjacent neighbors, the Inter Cluster synchronization uses the same approach as the Intra Cluster Synchronization uses. Therefore, the multi-hop synchronization precision Π_{inter} for tightly synchronized nodes can be bounded by using the hop count h:

$$\pi_{inter} \leqslant h(2+\mathrm{OD})\Delta t_b(\rho(2+\tau)+k_r\tau) \tag{4.69}$$

For loosely synchronized nodes the offset of the last synchronization will be a relevant issue. However, this is only a problem in mobile systems, since static systems will always compensate the offset as long as the time between synchronizations is not bigger than the maximum considered offset. A direct consequence is that the synchronization precision in multi-hop scenarios can be greatly enhanced if routes of tightly synchronized nodes are chosen. However, such optimizations exceed the scope of this thesis.

Mobility

Mobility influences the networks topology and therefore, the association of *Slaves* to *Master*s and the interconnectivity between *Masters*. Consequently, if a *Slave* loses connection to is respective *Master*, its offset towards the *Cluster* increases as no further compensation of the clock drift happens. If communication is re-established with the same *Master*, the node's maximum offset $o_{r,s}(t)$ will depend on the time between loss of link and the reception of the next beacon Δt and the offset of *Master* and *Slave* at the last synchronization as described by (4.67). A tightly synchronized *Slave* will increase its offset based on the time of the last synchronization and the drift of the nodes. A loosely synchronized node will additionally increase the offset by a value proportional to last offset ($o_{r,s}(t_{r,i})$), time since last synchronization (Δt), drift (ρ) and network tightness (τ). Depending on the value of k_r fast moving *Slaves* will never reach a tightly synchronized state, because they are switching *Master* faster than the algorithm is able to compensate the initial offset. On the other hand, a high k_r will enforce the induced errors when in orphaned state and the *Slave* reconnects to the same *Master*.

In contrast to the independence of the movement of *Slaves*, the movement of *Masters* has an impact on the whole PAN created by this *Master*. Therefore, the *Master* should be selected based on their topological speed, representing the amount of changes in their topology. Nodes with slowly changing topologies are better suited as *Masters*, as they increase the time of the algorithm to converge the offset towards zero.

The Inter Cluster Synchronization handles mobility well, since the mobility of a node in the local neighborhood of P_{c_1} does not change P_{c_1}'s virtual clocks of the adjacent nodes. Therefore, the transformation of the events is independent of each other. Since there is no hierarchy between the *Masters*, the movement of each node only effects the offset estimation towards its neighbors and the estimation of the neighbors towards this node. New nodes in a neighborhood start with an infinite uncertainty in the offset estimation since they have not yet established an offset estimation towards their neighbors. While the nodes receive beacons from adjacent *Masters*, they improve the offset estimation and decrease the uncertainty.

Estimating the Uncertainty

The estimation of the current uncertainty of the synchronization of the virtual clocks is difficult. Multiple factors influence the actual uncertainty in the synchronization, like beacon losses and the current drift of the individual clocks. In this approach the synchronization error $\epsilon_{r,s,i}$ of synchronization round i between two adjacent *Master*s P_{c_s} and P_{c_r} is characterized by their offset $\delta_{r,s,i+1}$ at beginning of synchronization round $i+1$.

Following PCS, the synchronization error is modelled as a zero-mean Gaussian distribution $N(0,\sigma)_{r,s}$. To estimate the standard deviation, the synchronization errors of the previous n synchronization rounds are used as sample set $E_{r,s} = \{\epsilon_{i-n}, \epsilon_{i-n+1} \dots \epsilon_i\}$. The standard deviation $\sigma_{r,s}$ of the zero-mean Gaussian distribution is estimated based on the sample set. The confidence interval $\left[\bar{x} \pm z_{\left(\frac{1+\gamma}{2}\right)} \frac{\sigma}{\sqrt{n}}\right]$ of the synchronization with typical probability γ can now be computed. $z_{\left(\frac{1+\gamma}{2}\right)}$ represents the $\frac{1+\gamma}{2}$-quantile of the standardized Gaussian distribution. The resulting size of the confidence interval $\alpha_{r,s} = z_{\left(\frac{1+\gamma}{2}\right)} \frac{\sigma}{\sqrt{n}}$ represents the current uncertainty estimation, which is incorporated in the *Time Attribute Reference* and consequently in the results of the *Transformation.*

The complexity of this computation is only dependent on n, which represents a trade-off between estimation accuracy and memory and computation overhead. The quantile of the standardized normal distribution is a pre-defined constant, characterizing the trade-off between accuracy of the estimation and growth of the uncertainty of the transformed events.

4.6. Abstract Sensor Event Transformation Engine

The Abstract Sensor Event Transformation Engine (ASETE) is responsible to find, combine and execute existing *Transformations* as necessary to automatically couple publishers and subscribers. To this end, multiple sub-tasks need to be solved. Firstly, the existing transformations need to be represented and stored in a machine readable way. Secondly, on announcement of a publisher or subscription of a subscriber, necessary *Transformations* need to be automatically selected. Thirdly, the transformations need to be executed on reception of an ASE *compatible* to the *Composite Transformation.*

The generic structure of the transformations enable an easy description of the *Transformation* by using the EID of the *Event Scheme* and the *Format Hash* of the *Event Format* as input and output data type, as described in Section 4.4.7 and 4.4.8. This information is stored in ASETEs Knowledge Base (KB) of existing transformations. Additionally, the KB needs to store the currently announced publications of the system with their EID and *Format Hash* and the established *Channels*. The existing transformations need to be handled differently depending on the following parameters:

Generality : General transforms are templates which automatically adept themselves to the input *Event Types*. The adaptation is based on the EID and *Format Hash* of the input events. This enables a more general description of the transformation behavior and eases the system specification. Consequently, they need to be adapted to the supplied input and output *Event Types* on run time. These *Transformations* are programs, which are executed if a channel is established and output a *Configured Transformation* matching the channel.

Homogenity : Homogeneous transforms are transparent regarding the *Event Scheme*. Therefore, they need to be applied to each publisher or subscriber depending on the currently considered events and filter expressions. On channel creation they need to be considered as soon as a filter expression exists, either specified by the subscriber or by a *Transformation* in the *Composite Transformation DAG*.

In the following sections, the selection, creation and application mechanisms of the ASETE will be shown.

4.6.1. Automatic Configuration of Generic Transforms

Existing *Generic Transformations* such as *Attribute Transformations* or *Inter-* or *Extrapolation Transformations* need to be fully specified on run-time. These transformations are not specific to any phenomenon, but act directly on attributes or whole events. Consequently, they are specified in the KB with the special output EID `any`. In general, these *Transformations* are supplied with the list of *Input Event Types* ${}^{T}e_{in}$ and the desired *Output Event Type* ${}^{T}E_{goal}$ and the attached filter expressions P to enable configuration. In the following the necessary configuration mechanisms for the *Generic Transformations*, see Sections 4.5.2, 4.5.5 and 4.5.6, are described.

Cast Transformations are *Unary Transformations* and have only a single *Input Event Type* ${}^{T}e_{in}[0]$ and an empty set of filter predicates $P = \varnothing$. The configurations of these transformations create an associative mapping from AID to the goal data type $m = \left\{(a_i^T, a_j^T) | a_i \in^T e_{in}[0], a_j^T \in^T e_{goal}, a_i^{id} = a_j^{id}\right\}$. These are executed on reception of the event as a cast operation $e_{out}.a_j = \left(m[a_i^{id}]\right) e_0.a_i$ for each attribute contained in the event.

Rescale Transformations are similar to *Cast Transformations*, but instead of transforming the data type, they transform the scaling of linear scales of attributes. On creation of the channel they create an associative map of attribute scale quotients $m = \left\{(a_i^{id}, \frac{a_i^s}{a_j^s}) | a_i \in^T e_{in}[0], a_j^T \in^T e_{goal}, a_i^{id} = a_j^{id}\right\}$. On reception of an event they operate on the value and the scale of the attributes according to Equations 4.70 and 4.71.

$$a_j^s = m[a_i^{id}]a_i^s \tag{4.70}$$
$$a_j^v = m[a_i^{id}]^{-1}a_i \tag{4.71}$$

Scale Change Transformations are more complex than *Cast* and *Rescale Transformation*, since they need additional external information to modify attributes of incoming ASEs. They are *Dual Transformations* by using an input event e_0 and a reference event e_{ref}. The goal of *Scale Change Transformations* is the change of reference of the attributes value or the switch to a completely new scaling system. An example is the transformation from a local time value of a sensor node to an Unix time stamp. Similar to the previous *Transformations* on channel creation, an associative set is created, which contains tuples of AID and the scales of output and input $m = \{(a_i^{id}, (a_i^s, a_j^s)|\}$. On reception of a **Reference** event, the event is stored in a buffer to provide the necessary $\Delta ri, j$. On reception of an input event the scale change is executed as described by Equation 4.30 of Section 4.4.3.

Generic Event Transformations are typically homogeneous transformations combining multiple similar events to output events of the same type, but change context parameters. Consequently, they need to be configured on channel creation to the *Event Type* of the input event, even though they may be applied dynamically on every incoming event. This adaptation consists of the extraction of the *Attribute Types* and the selection of the operation executed on them. Two different types of attributes need to be handled: *Continuous Attributes* and *Discrete Attributes.* The configuration will assign the generic operations to the different AIDs according to the definition of the *Transformation.* The *Attributes* the *Transformation* acts on are selected based on the *Filter Expression.* The remaining *Attributes* act as internal *Filter Expression* to prevent incompatible ASEs to be fused. *Continuous Attributes* need to be equal as stated by the definition of equality for continuous uncertainty values in Section 4.4.4. *Discrete Attributes* need to be equal with the resulting uncertainty be < 1.

4.6.2. Representing Rules as Knowledge Graph

The existing *Transformations* form complex interactions as multiple individual transformations may be necessary to fulfill the subscription of a CPS component. An example is a system containing publishers of *Position Information*, but CPS components require acceleration information. To transform the existing *Position Information* in *Acceleration Information*, two differentiation steps are necessary. Both operations can be described as individual *Complementary Fusion Transformations.* However, a single stage lookup on the transformations will not find any transformation enabling an inference of acceleration from positions. Therefore, an extended lookup system needs to be used to find *Complex Transformations* composed of multiple elementary transformations.

The base of this extended lookup is the representation of all *non-Generic Fusion Transformations* in a directed graph. This graph connects each transformation with its outgoing and incoming *Event Schemes*. *Generic Transformations* cannot be included in the graph as they have unknown input and output EIDs until configured. The result of such a representation for an exemplary set of transformations is shown in Figure 4.28. As visible, the graph contains cycles, as *Event Schemes* exist that are input and output to some subset of the transformations. An example is the transformation of two positions to a proximity sensor event and the back-transformation to a position event. Theoretically, this may create an infinite amount of events, because results are fed back as input. Therefore, the used lookup algorithm needs to only consider transformations decreasing the distance of the input *Event Schemes* to the target *Event Schemes*.

Another property of the representation of the *Transformations* as a graph is the separation of disjunct sets of transformations. If two sets of transformations share no *Event Scheme* in the input and output specification of their transformations, they represent separate subgraphs. This enables a faster lookup of the necessary transformations, as only a single one of these need to be considered.

The graph can be used to formulate the lookup of the necessary transformations as an algorithm on graphs. In the case of a subscriber for an arbitrary *Event Scheme* in the graph, this *Event Scheme* is used as the root node of a *Spanning Tree*. The resulting tree contains all applicable transformations, that might lead to the desired output *Event Scheme*. The exact algorithm used is described in the next Section 4.6.3.

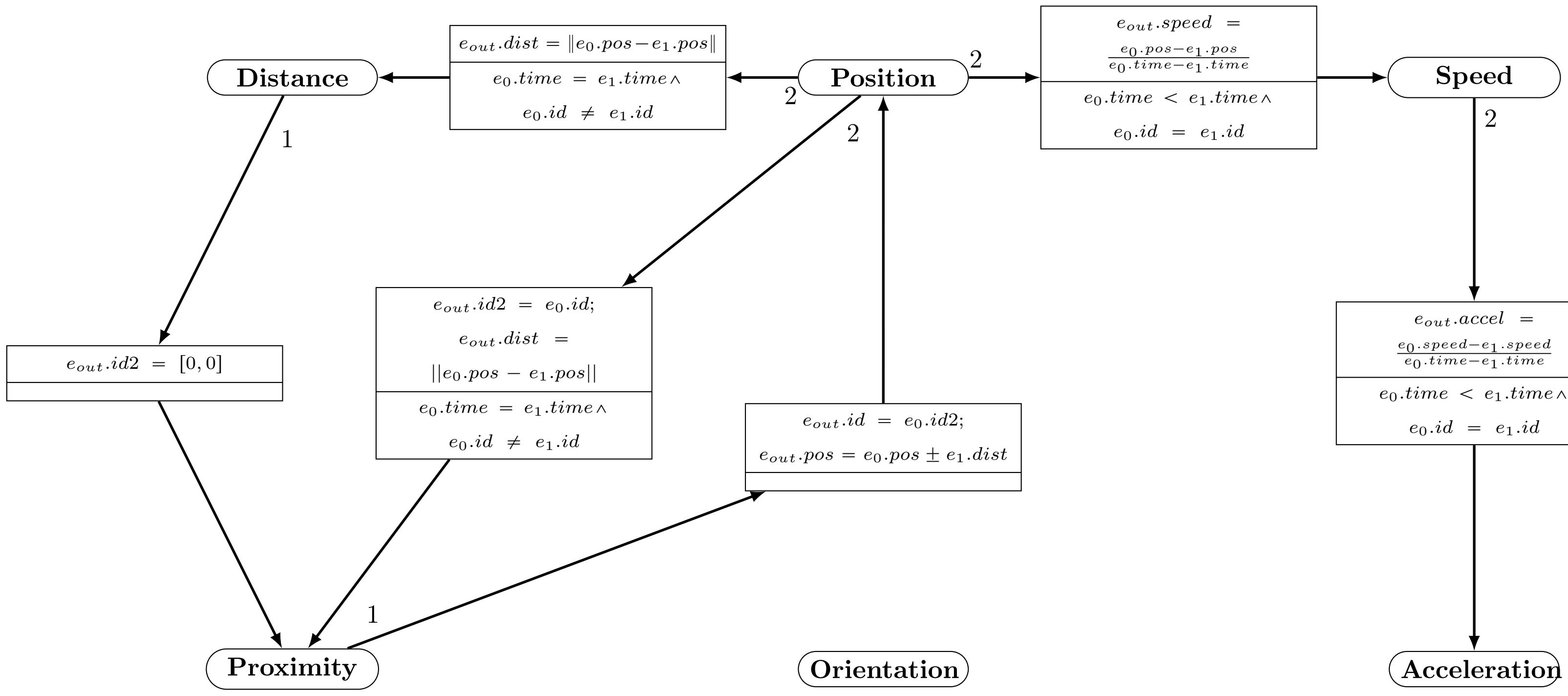

Distance
$e_{out}.dist = \|e_0.pos - e_1.pos\|$
$e_0.time = e_1.time \wedge$
$e_0.id \neq e_1.id$
Position
2
2
2
$e_{out}.speed = \frac{e_0.pos - e_1.pos}{e_0.time - e_1.time}$
$e_0.time < e_1.time \wedge$
$e_0.id = e_1.id$
Speed
2
1
$e_{out}.id2 = [0, 0]$
$e_{out}.id2 = e_0.id;$
$e_{out}.dist = ||e_0.pos - e_1.pos||$
$e_0.time = e_1.time \wedge$
$e_0.id \neq e_1.id$
$e_{out}.id = e_0.id2;$
$e_{out}.pos = e_0.pos \pm e_1.dist$
$e_{out}.accel = \frac{e_0.speed - e_1.speed}{e_0.time - e_1.time}$
$e_0.time < e_1.time \wedge$
$e_0.id = e_1.id$
1
Proximity
Orientation
Acceleration

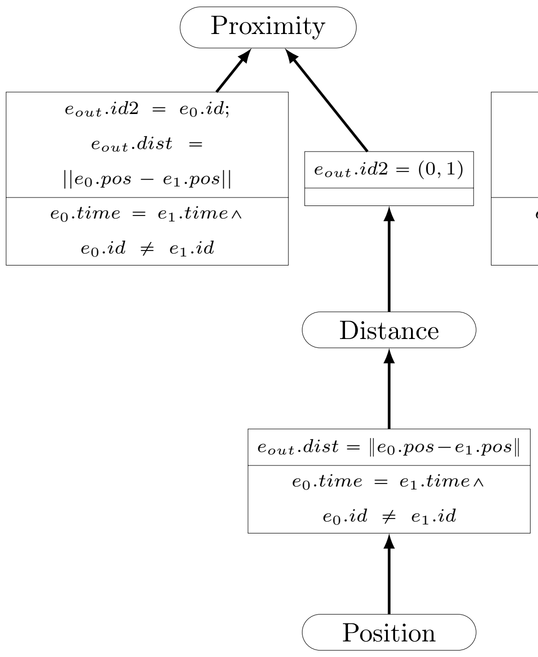
Figure 4.28.: An example knowledge graph consisting of relevant transformations concerning physical properties of mobile entities.

4.6.3. ASETE Channel Creation

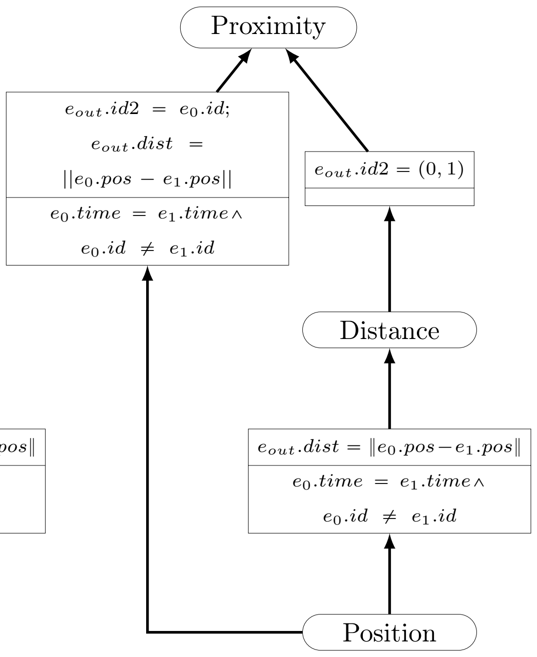
Algorithm 1 Algorithm to extract the *Transformation DAG* for *Event Scheme s* from the *Knowledge Base kb* and the current published *Event Scheme* list *pList*.

```
procedure CREATEDAG($s, kb, pList$)
    $t \leftarrow spanningTree(s, kb)$                    ▷ create spanning tree of $kb$ with root node $s$
    for $n$ in $t$ do                                     ▷ Iterate the nodes of the DAG
        if $n.type = Transformation$ then                 ▷ handle transformations
            $pruned \leftarrow false$
            for $s_{in}$ in $n.input$ do                  ▷ check incoming event schemes
                if $s_{in}$ not in $pList$ then           ▷ check for existing publisher
                    PRUNE($t, n$)                         ▷ prune sub-tree
                    $pruned \leftarrow true$
                end if
                if not $pruned \wedge e(s_{in}, n)$ not in $t$ then    ▷ check consistency
                    INSERT($e(s_{in}, n), t$)             ▷ Restore consistency
                end if
            end for
        end if
    end for
end procedure
```

The ASEIA broker handles the registration of publishers and subscribers in the system. On each received publisher announcement the EID and *Format Hash* of the *Event Type* is added to the *Knowledge Base*. On each received subscription ASETE checks the *Transformations* stored to create an *ASEIA-Channel*. To generate the *Composite Transformations*, a complex search needs to be executed to generate a *Transformation DAG* stored in a *Composite Transformation*. The first step is the generation of the *Transformation DAG* of non-generic non-homogeneous *Transformations*. This process is described by Algorithm 1. To select applicable *Complementary Fusion Transformations* from the *Knowledge Base*, the graph representation of the knowledge base can be used. As described in Section 4.6.2, the graph representation enables the creation of spanning trees by using the subscribers *Event Scheme* as root node. The spanning tree should be as balanced as possible, since a balanced tree minimizes the maximum distance of the transformations from the root node. This helps to minimize the additionally generated uncertainty, because each transformation on induce may induce uncertainty. Afterwards, the tree is not a valid transformation DAG, since connections between input *Event Schemes* and *Transformations* are removed as the tree is constructed, as shown in Figure 4.29a. However, a *Composite Transformation* is only valid if all its incoming *Event Schemes* are connected and published by at least one publisher. Consequently, the tree will be filled with the missing connections based on the *Transformation's* input *Event Types*. The repaired *Transformation DAG* of the example is shown in Figure 4.29b. After the creation of the *Transformation DAG* compatibility between input *Event Types* and published *Event Types* has still not been established. The *Event Formats* might be different for the individual input *Event Schemes* of the *Composite transformation*. The *Generic Attribute Transformations* are now configured to match the



(a) An example Spanning Tree created by the system for the vehicle scenario for a *Proximity Subscriber*.



(b) An example Transformation DAG created by the system for the vehicle scenario for a *Proximity Subscriber*.

different *Event Formats* of the input *Event Schemes*. To this end, the *N-Ary Transformations* are searched and configured. Afterwards, the *Unary Attribute Transformations* are searched and configured to finally match the *Event Types*. If no match can be found the part of the *Transformation DAG* need to be pruned until an *Event Scheme* is reached, which is fulfilled either by a publisher or by another sub-DAG. The process of configuring *Generic Transformations* is described in Section 4.6.1.

On generation of the *Composite Transformation* from the *Transformation DAG* a buffer needs to be created for each contained Transformation. The buffer handles the different timing of the events received by the broker and serves as storage for multiple input *Transformations*, this simplifies the API as described in Section 5.2.3. Multiple *Buffer Types* exist to provide the different types of *Transformations* with an appropriate buffering mechanism, which may also be configured. The general *Buffer Types* are described in the next section.

4.6.4. Buffering Incoming Events

Event-based systems allow no prediction of the arrival order of input ASEs. Therefore, it is necessary to buffer them to provide consistent input event sets to the *Transformations*. Consequently, each transformation is accompanied with a buffering structure to temporarily store incoming events. This is similar to the concept of Aurora (see Section 3.5, which uses temporary buffers to store windows of the input stream. The structure and the behavior of these buffers depend on the type of the transformation. There exist five types of buffers depending on the transformations:

Unary Transformation Buffers are pseudo-buffers, that do not store anything. The input events are directly forwarded to the transformation for execution, since no other events are necessary.

Reference Transformation Buffers are one element buffers, that store reference information used in *Generic Transformation*. These transformations only need the most recent information. Therefore, only the reference with the smallest age is kept in the buffer. This assumption is reasonable, as the uncertainty of reference points should be smaller then their change over time.

N-ary Transformation Buffers are generic buffers enabling separate storage of different types of events. They provide the facility to automatically generate combinations of stored input events and deliver them to the transformation. The transformation needs to check the passed input event tuples for usability and generates output event sets accordingly.

Associative N-Ary Buffers are generic buffers, which separate the incoming events into multiple lists based on a specified *Attribute*. They are used to enhance performance, as they limit the amount generated event combinations because of early filtering.

Custom Buffers are user implementable buffers enabling arbitrary ordering and combination generation.

The behavior of the buffers is very important to the performance of the *Transformation* execution, as all transformations, which are not *Attribute Transformations* or *Unary*, need to generate combinations of input events to form tuples. The generation of the tuples is handled by the buffers and depend on the following configuration parameters, which consequently influence execution performance heavily:

Maximum Size configures the maximum count of input events stored for each input event type. The parameter is only applicable to *N-ary Transformation Buffers* or *Custom Buffers*.

Maximum Age configures the timeout after which an event is removed from the input event storage. The time reference is always the current time of the broker the buffer is executed. The age is computed based on the time attribute of each input event and the current time of the *ASEIA Broker Node*.

Maximum Uncertainty is a scalar value applied to the sum of uncertainty of all *Attributes* which are used by the *Transformation*. Input events, whose uncertainty surpass this value, will not be added to the input event storage. To compute the uncertainty of an attribute, the norm of the uncertainty vector or matrix is used.

The *Maximum Size* parameter n is the most important one as it directly defines the worst case complexity of the transformation execution as $\mathcal{O}\left(n^m\right)$, with m being the arity of the transformation. In worst case all input event combinations need to be check regarding the

Filter Expression of the *Transformation* or the subscriber. In general, the actual effort is much lower as static predicates lessen the necessary comparisons heavily. The impact of *Maximum Age* and *Maximum Uncertainty* depend on the supplied ASEs. In systems with very uncertain information or very long periods these parameters can greatly improve the performance of the system.

The parameters are either configured on design time or passed as part of the *Policy*-System, which is described in Section 4.6.5.

4.6.5. Activating Transformations

The activation of a *Composite Transformation* of a channel is executed by Algorithm 2.

Algorithm 2 Algorithm feeding an input ASE e_{in} to a *Transformation DAG tG* to produce a set of output ASEs $E_{out} = \{e_{out0}, .., e_{outn}\}$.

```
 1: procedure CALL($tG, e_{in}$)                      ▷ Execution of the whole Transformation DAG
 2:     $e_{out} \leftarrow \emptyset$
 3:     for $t \in tG$ do                                          ▷ Iterate over all Transformations
 4:         if accept($t, e_{in}$) then               ▷ Check if Transformation accepts $e_{in}$ as input
 5:             $e_{out} \leftarrow e_{out}$+callTransformation($t, e_{in}$)             ▷ Append all results
 6:         end if
 7:     end for
 8:     return $e_{out}$
 9: end procedure

10: procedure CALLTRANSFORMATION($t, e_{in}$)                ▷ Execute a single Transformation
11:     $e_{temp} \leftarrow t(e_{in})$
12:     $T \leftarrow$linked($t$)                                  ▷ Get all linked transformations
13:     if $T == \emptyset$ then
14:         return $e_{temp}$                                          ▷ Pass results to $e_{out}$
15:     else
16:         $e_{out} \leftarrow \emptyset$
17:         for $t_{next} \in T$ do                                  ▷ Forward output inside DAG
18:             for $e \in e_{temp}$ do                                 ▷ Forward all created events
19:                 $e_{out} \leftarrow e_{out}$+callTransformation($t_{next}, e$)         ▷ Recurse until root
20:             end for
21:         end for
22:         return $e_{out}$
23:     end if
24: end procedure
```

Transformation DAGs, as part of established channels, are activated whenever a compatible ASE is received by the broker. In this case *compatibility* is defined according to the *Event Type* declared in the input event list of the channel. For an ASE e_{in} and an *Event Type* ${}^{T}e_0$ to be compatible, all *Attribute Types* ${}^{T}a_{0,i}$ contained in ${}^{T}e_0$ need to be compatible to the corresponding *Attribute Types* ${}^{T}a_{in,i}$ of the event. Two *Attribute Types* are considered to be compatible when they are equal. In consequence, a channel may also accept events, which contain more information than needed by the contained *Transformations*. This is to be expected as the *Event IDs* of the *Event Type* and the ASE show the same behavior, see Section 4.4.7.

The contained *Transformation DAG* forwards the received event e_0 to the input buffers of all leaf transformations $t_{0,i}$ through the execution operation of the individual *Transformation*. The output of the transformation e_{0,i_j}, if there is any, will then be put in the next transformation's $t_{1,j}$ input buffer through the execution operation of this transformation. This process continues unless no output events are created for the next layer. If the transformation is the root transformation of the *Transformation DAG*, the resulting events will be published according to the specified execution policy.

Execution *Policies* enable the subscriber to influence the amount of published output events based on their properties. These policies will act as a filter on the output events and may change the behavior of the input buffers of the transformations. The subscribers state the required policy through a filter statement on a virtual *Policy* attribute. This attribute is not contained in neither the input nor the output events and is only used to provide a concise API for selecting policies.

The generic *Policies* of ASEIA are:

All is the default *Policy*. It outputs all events, which fulfill the subscriber's requirements stated in the filter expression passed on channel creation. It may create a large amount of output events depending on the size of the *Transformation DAG*, the buffer size of the contained transformations and the conditioning of the events. However, the amount of events is never infinite unless the resulting *Transformation* is homogeneous and no recursion preventing filter expression is specified.

Newest is a *Policy* aiming to provide output events with the smallest age, which means the difference between the time stamp of the output event and the current time of the executing broker shall be minimal. This policy changes the behavior of the underlying buffers of the individual transformations by sorting the buffer according to the time attribute of the events. The transformations are executed from newest to oldest events combinations. The execution of the *Transformation DAG* stops after the first event is created that fulfills the requirements of the subscriber.

Best is a *Policy* aiming to minimize output of *Transformation DAGs* by sorting the output events according to their uncertainty. The general approach is to compute for each attribute the norm of the uncertainty and sum these over all attributes. The policy will discard events which are equal according to: $\forall a_0 \in e_0, \forall a_1 \in e_1 : a_0.id == a_1.id \rightarrow a_0 == a_1$ and have larger uncertainty. Since the amount of output events is unknown on design time, this operation needs an additional output buffer in the created channel, which can grow dynamically. The comparison algorithm has a worst case complexity of $\mathcal{O}(n^2)$, where n is the amount of created events. This policy allows to trade computation against communication with minimum impact on the uncertainty of the generated events.

4.7. Summary

This chapter describes the concepts of the ASEIA framework. The framework uses a generic context description mechanism to represent sensor information in an abstract way and communicate them using events in a distributed system. The goal of the extensions is a dynamic composition of CPS, which consists of multiple independently developed components, whose context requirements on the necessary sensor information is incompatible. It enables filtering, publication and subscription of these events based on the contextual and semantic information contained in the events with a focus on the applicability to low-power embedded systems. The necessary information processing to translate the context of the information between the CPS' components is described by generic *Transformations*, which are enabled automatically. The system infers necessary *Transformation DAGs* on run time to fulfill subscriber's requirements. The *Transformations* are categorized and described with their properties in a hierarchy together with their adaptation mechanism used to decouple the CPS components. Finally, the chapter describes the integration of the description and transformation system to a generic P/S system. This enables an extension of most existing systems with the additional functionality and paves the way for generic dynamic composable and adaptive CPS.

5. Implementation

The following chapter describes the implementation of ASEIA, described in Chapter 4, and its adaption to an actual P/S system.

5.1. ASETE Implementation and Language Bindings

The implementation of ASEIA's concepts is divided into three major blocks:

ASETE Transformation Core consists of the implementation of the *Knowledge Base*, the *Transformation DAG* generator and the *Transformation DAG Execution Engine*. The *Knowledge Base* needs to be able to efficiently store and query existing *Transformations*. The *Transformation DAG Generator* enables the composition of existing *Transformations* into DAGs based on dynamically specified requirements of the subscribers regarding *Event Scheme*, *Event Format*, QoS, QoC and context. The *Transformation DAG Execution Engine* executes the generated DAGs on incoming ASEs. It is the most performance critical component of ASETE, as it is executed more often then the others.

Transformations are either specified as *Generic Transformations*, which adapt themselves to the requirements of the subscribers on *Transformation DAG* generation or as *Heterogeneous Transformation*, which needs to be specified based on the application scenario. Both types benefit from a homogeneous abstraction providing linear algebra operations on ASE attributes incorporating values, uncertainty and context. This abstraction may either be provided as a programming API or in the form of a DSL.

CPS API provides the binding of the CPS applications to ASEIA's automatic transformations and description mechanisms. The API should be designed to enable different language bindings. This minimizes the effort of developers and designers to use ASEIA in existing CPS components. The API shall provide mechanisms to specify *Event Schemes* and *Event Formats*, *Filter Expressions* and linear algebra operations directly on the received ASEs. The specification of *Event Schemes*, *Event Formats* and *Filter Expressions* as DSLs enables an extensive error checking and optimization, which is important for embedded systems.

These blocks provide logical separation in the implementation, which enables different languages to be chosen for the individual blocks. This increases ASEIAs flexibility regarding the integration into existing CPS frameworks. However, some requirements regarding features of the used languages exist, which are induced by the conceptual design of the ASEIA components. The following sections will discuss possible languages usable to implement the blocks and the actual choice used for the prototype.

5.1.1. ASETE Implementation Language

The ASETE implementation language can be chosen independently of the other components and is not visible to either the *Transformations* nor the *CPS API*. Additionally, it shares no common dependent internal components with the other parts of ASEIA. Consequently, a language can be chosen, which provides maximum efficiency regarding implementation effort, storage efficiency and execution efficiency. In the following, the prototype of ASEIA is implemented using *C++*, as it provides highly optimized code for many architectures and many libraries implementing necessary algorithms such as the *STL* [103] for generic algorithms regarding lists, heaps, hashes and more. Additionally, the *Boost Graph Library* provides flexible, high efficiency algorithms on arbitrary graphs easing the implementation of the *Knowledge Base* and the generation of *Transformation DAGs*.

5.1.2. Cyber-Physical-System (CPS) Application Bindings

Language bindings are necessary to enable CPS components to use ASEIA's advanced concepts. These bindings enable an CPS application to subscribe and publish any ASEs. To this end, the ASEs need to be described. In the case of subscriptions, the requirements of the CPS component regarding the ASE need to be specified too. The language binding needs to provide an API, which provides access to publication and subscription of generic ASEs together with the capability to specify the requirements in case of subscriptions. Additionally, the algebraic operations defined on the ASEs shall be accessible to the CPS component natively. This enables a direct integration of th received data in the local processing within the component. ASEIA assumes some capabilities of the languages it binds to. ASEIA considers object-oriented languages only, as the ASEs are defined as an *Event Hierarchy*, which minimizes necessary definitions and automatically enables conversion of ASE towards the root of the hierarchy as defined in Section 4.4.6. CPS publish and subscribe exactly defined ASEs with fully specified *Event Schemes* and *Event Formats*. Consequently, statically typed languages enable additional checks on compile-time, which may be used to detect programming errors early, e.g incompatibility between ASE definition and supplied ASE to an ASEIA publisher. Additionally, these languages may optimize the code to minimize resource consumption and increase performance.

Each language binding consists of three parts:

ASE Specification DSL enables the composition of ASE from existing ASE and custom attributes. It provides the specification infrastructure towards the *Event Schemes* and *Event Formats*.

Filter Expressions DSL enables the specification of *Static* and *Dynamic Selection* criteria, which can be attached to ASE subscriptions.

ASE P/S API enables publication and subscription of ASE described by the *ASE Specification DSL* and the *Filter Expressions DSL*. It implements mechanisms to access *Attributes* contained in the ASE and type information regarding the channel static *Event Scheme* and *Event Format*. Additionally, it provides access to the algebraic operations on ASEs.

The following paragraphs discuss possibilities to implement these three parts for three reference languages:

***C++* [158]** is a well established language, which has been existing for more than 20 years. It provides strict type safety, checked on compile time, with the ability to heavily optimize the resulting binary code. *C++* supports multiple programming abstractions, ranging from *Object-Orientation* over *Functional* to *Generic Programming.* The *C++* TMP language is an especially powerful tool to enhance the performance of applications, as it allows offloading static checks and computations to the compiler [19]. TMP can be used to implement DSLs directly in *C++* without any additional tools, as described and used by Czarnecki et al. [54]. The integration of the DSL in the *C++* compile process enables automatic optimization of the DSL expressions. *C++* does not need any run time infrastructure, which eases the deployment to deeply embedded systems. It also enables highly flexible APIs without run time overhead, see [154].

***Java* [73]** was developed to be compiled once and shipped to every capable device. It is supported on a wide range of architectures, but generally needs operating system support to enable its Virtual Machine (VM). However, there are exceptions such as special *ARM* architectures that provide a streamlined VM in hardware. *Java* provides many programming abstractions as core concepts of the language. It enables object-oriented, functional and stream programming. The *Java* class library implements solutions for most generic problems and a huge set of libraries exists. As *Java* is a strictly typed language, type checks are executed on compile time and only cast operations may fail on run time. The performance of *Java* is generally worse compared to directly to machine code compiled languages, as certain optimizations are not possible on compile time [164]. Some VMs, such as the *HotSpot* VM [120], exist that solve this issue with dynamic optimization on run time on non-embedded system. However, limited VMs exist which enable *Java* on embedded systems such as the *SQUAWK* VM [148]. DSLs can be implemented using the *Java Compiler Compiler (javacc)* [88], which generates a language parser from a DSL specification. The resulting parsers provide native interfaces to integrate them in any *Java* program. This may be used to implement the *ASE Specification DSL* and the *Filter Expression DSL.*

Python is a scripting language focusing on a simplistic language with a huge library solving the most common generic problems [161]. Even though the language is designed for scientific analysis and operating system scripting, there are ports of the interpreter to embedded systems. *MicroPython* [71] enables support for low power AVR architectures, but with limited functionality. *Python* enables many programming abstractions such as *Object-Orientation*, functional and generic programming. A positive aspect is the easy integration of *C/C++* libraries in *Python*, which largely extends the available libraries. The drawback of *Python* is similar to *Java*, optimization is very limited and the generic object orientation support enables some run time overhead. In contrast to *C++* and *Java*, the types of *Python* data can only be checked on run time. Therefore, type errors are recognized late. Additionally, *Python* only evaluates the code paths, which are currently executed. Consequently, bugs in seldom used code paths are difficult to detect. The interpretation of the specified programs enable the implementation of arbitrary DSLs as a translation to native *Python* code and its execution.

Result ASEIA's prototype applications will use a *C++* language binding because of the minimal performance impact and the strict error checks. This eases the evaluation of ASEIA's performance regarding CPU and memory consumption. Additionally, it provides the maximum amount of checks regarding implementation errors in the CPS components, which will be discussed in Chapter 6.

5.1.3. Transformation Implementation

The *Transformations* need to be implemented differently from the CPS API because each *Transformation* needs to be at least adaptive to changes in the *Event Format*. Consequently, strict type checks at compile time are impossible. *Generic Transformations* need to be adaptive regarding the *Event Scheme* and the specified requirements of the subscribers. Consequently, they are small programs, which output *Configured Transformations*. These are then included in the final *Transformation DAG*. Similar to the CPS API multiple language bindings are possible for the implementation of *Transformations*. The requirements are slightly different in this case. Languages providing object orientation are beneficial to this case, as the *Transformation Hierarchy* provides different *Abstract Transformation Classes*, which ease the programming effort. Additionally, languages, which allow operator overloading enable a native use of mathematical operators to specify the necessary operations executed on the ASE's attributes. To implement these operation the used language should provide a high-performance linear algebra library. *Transformations* are executed often, which mandates a language enabling optimization to decrease the load of the broker nodes executing them. Function shipping is a very beneficial property as it provides the capability to deliver additional *Transformations* on run-time to the broker nodes of the system. Another approach is the dynamic evaluation of the *Transformations*, which induces large run-time overheads as no static optimization is possible. Consequently, *Transformation* implementation is possible in all three languages considered in Section 5.1.2. *C++* provides run-time performance and operator overloading. The *Eigen* [74], [75] library provides fast and flexible linear algebra operations usable to implement the mathematical operations on ASE's attributes. *Java* provides static and run-time optimization and function shipping and enables linear algebra operations using *la4j* [91]. *Python* provides dynamic evaluation and function shipping enabling maximum flexibility in the *Transformation* specification. *Numpy* [116] provides a huge library of linear algebra operations and data types for *Python*. Consequently, all three languages are viable candidates to implement the *Transformation* API.

The prototype of ASEIA will use *C++* to implement *Transformations* because of its inherent optimization and run-time benefits. This enables the evaluation to test the performance of the generic concept and minimizes the influence of the language and its abstractions. However, the other languages can be supported similarly, by using the same implementation design as specified in Section 5.2.3.

5.2. Implementation of the P/S Overlay

The implementation of ASEIA as an overlay to existing P/S systems needs a special software design regarding the structure and operations on ASEs to enable support for deeply embedded systems. The following Subsection 5.2.1 provides an overview of the used class structure to implement ASEs. The next Subsection 5.2.2 describes the ASE API presented to CPS applications. Afterwards, Subsection 5.2.3 describes the API used to implement the *Transformations* as operations on ASEs. Subsection 5.2.4 shows the design and implementation of the ASEIA KB by using *C++* facilities. The last Subsection 5.2.5 describes the necessary adapters to link ASEIA to the underlaying P/S system.

5.2.1. Overview of the Implementation

The implementation of ASEIA is based on ASEs. ASEs present the basic infrastructure to represent, transfer and process information in the system. The implementation is based on a class hierarchy consisting of three main parts. These parts are shown in Figure 5.1.

One part represents the ASE interface of CPS applications. This ASE class is called `Event` and uses TMP to configure the generic `Event` class with an application specified *Event Type*. To this end, the `Event` is composed of configured `Attributes`. The `Attributes`' configuration consists of the information specified in Section 4.4: ValueType `VT`, Scale `S`, Unit `Unit` and AttributeID `ID`. Similarly, the *Transformation* ASE interface consists of an ASE representing class called `MetaEvent`. This `MetaEvent` combines different `MetaAttributes` to a single ASE. In contrast to the `Event` class, this classes are configured on run time without generic programming. This enables the *Transformations* to be specified without exact knowledge of the *Event Format* and the exact *Event Scheme*. To handle the static *Event Type* data a third branch of the class hierarchy consists of the `EventType`, composed of `AttributeTypes`. This abstraction allows the comparison of types of `Events` and `MetaEvents` on run time. Additionally, it enables the configuration of `MetaEvents` on run time. There is no direct connection between `Events` and `MetaEvents`. Conversion is done through the serialization mechanism used to transport ASEs in the network. ASEIA uses a specialized serialization system instead of generic ones such as *boost serialize* [144]. It separates *Event Type* and *Event Data* serialization to optimize the amount of data needed to be transported. Additionally, it enables ASEIA to exchange *Event Type* data on channel creation independently from the serialization of *Event Data* after the channel is established. This functionality is realized by the (de-)serialization association between `Event`, `MetaEvent`, `EventType` and `PacketBuffer`.

The *Attribute* and *Attribute Type* representations follow the same three parts design as the *Event* representation. This structure is shown in Figure 5.2. The `Attributes` are configured by using discrete TMP parameters, configuring the *Attribute Type*. These define the implementation used for the four subparts of an `Attribute`. These parts are `Value`, `Scale`, `Unit` and `AttributeID`. `AttributeID`, `Scale` and `Unit` are pure representation of the *Event Type*. `Value` combines the *Event Type* with a run time storage of the *Event Data*. ASEIA represents all *Event Data* as matrices of uncertain values. These are implemented through the *Eigen* [74], [75] *C++* library for linear algebra. `AttributeTypes` are used to transfer this TMP parameters to a run time representation. This is necessary to create a homogeneous abstraction independent of the exact configuration of a CPS applications'

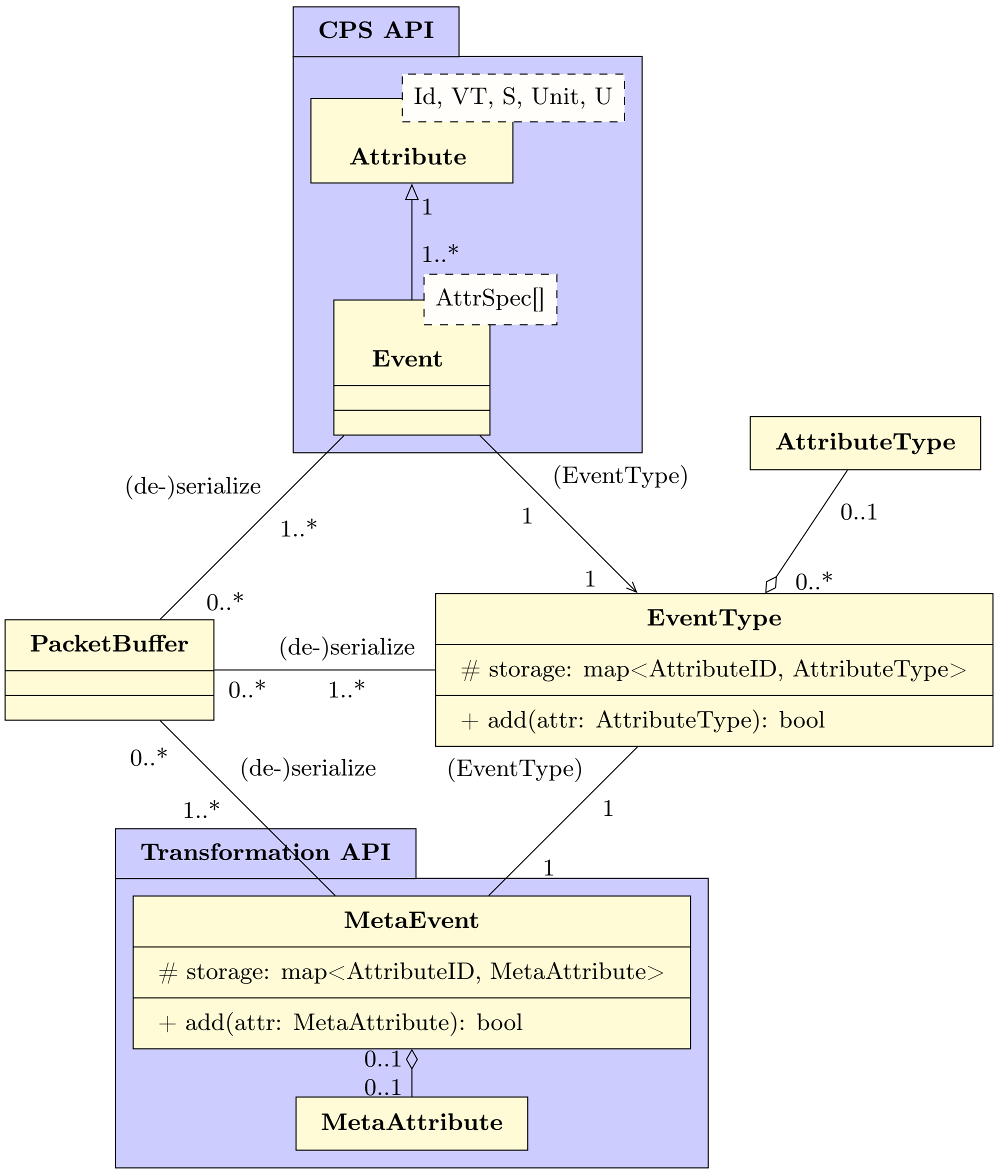


Figure 5.1.: Overview UML-Diagram showing the relations between the different abstractions used to represent ASEs.

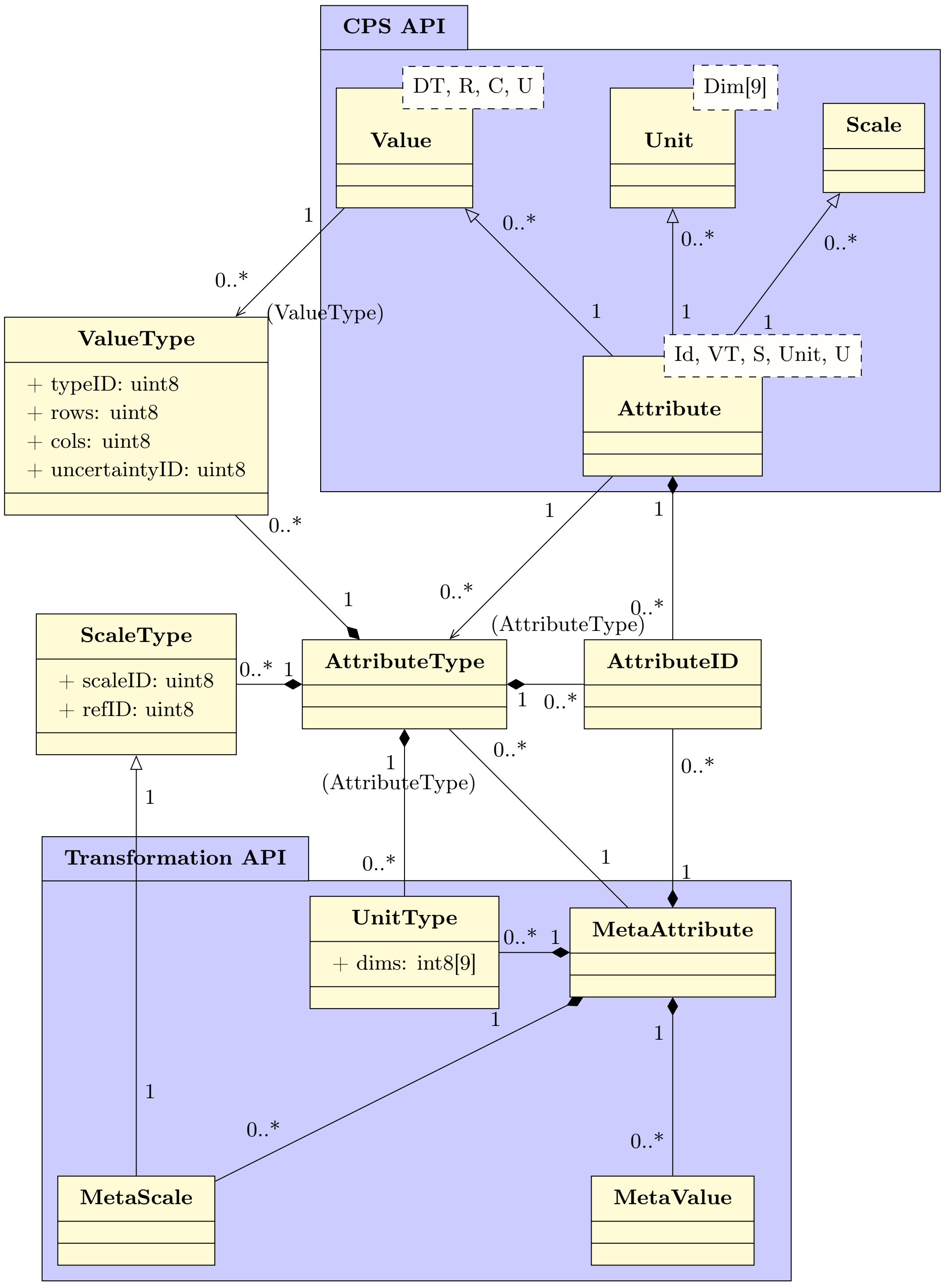


Figure 5.2.: UML-Diagram showing the different representations of *Attributes* of ASEs.

ASE. The `AttributeType`s also combine four information representation parts: `ValueType`, `ScaleType`, `UnitType` and `AttributeType`. `MetaAttribute`s contain the same *Event Data* and *Event Type* data as *Events*, but configured and represented on run time. Consequently, `MetaScale` and `MetaUnit` are extensions of `ScaleType` and `UnitType`. `AttributeID` is the same in all three representations and is part of the interface to extract a single *Attribute* from an ASE. The `MetaValue` implementation uses the object orientation feature of polymorphic behavior. This enables the actual value implementation to be chosen automatically on run time depending on the specified *Event Type*, Section 5.2.3 discusses this mechanism in detail.

The benefit of the separation of ASEIA into a dedicated CPS and a *Transformation* sub-system is the applicability to deeply embedded systems, as shown in [154]. The *Transformations* need considerably more memory and computational power to be executed than the pure CPS ASE communication facilities. Therefore, two separate APIs are defined, which are also implemented separately. The CPS API provides a maximum of compile time checks with minimum resource consumption, while the *Transformation* API provides maximum flexibility and dynamic extendability on run time to provide the necessary adaptation facilities.

5.2.2. CPS ASE API

The CPS ASE API is designed to provide maximum performance with maximum error reporting on compile time to perfectly suit deeply embedded systems. To this end, Template Meta-Programming (TMP) is used. TMP provides a functional language on top of *C++*, which enables the compiler to compute "classes". This technique is extensively described by Abrahams and Gurtovoy [19]. It is heavily used in the Standard Template Library (STL) [103] of *C++* to provide maximum performance with maximum flexibility by adapting the data structures and algorithms on compile time. The same mechanism can be used to adapt a generic `Event` class to different configurations. In this case, the `Event` is supplied a TMP vector, as provided by the boost MPL [76] library, containing the specification of each contained *AttributeID*. The compiler iterates the vector and creates appropriately configured `Attributes`. If an `AttributeID` exists twice in the vector, a compile time error is raised. The `Event` combines all generated `Attribute`s through multi-inheritance. The resulting `Event` "is" all of the `Attribute`s at the same time and allows the selection of a single `Attribute` by using the `attribute(AttributeID id)` method. In case no `Attribute` with the requested `AttributeID` is contained in the `Event` a compile time error is issued and the developer directly realizes his or her error. The benefit of this abstraction is the usage of the compiler to minimize run time overhead. The generated ASEs are guaranteed to only contain the *Event Data* as content, all *Event Type* information is encoded by the compiler in the generated *C++ Type*.

The specification of the ASEs forms a DSL representing a subset of *C++*. This DSL is shown in Listing 5.2 in Extended Backus-Naur Form (EBNF) form according to [3]. Additionally, Listing 5.1 provides basic definitions used also in *C++*. For the sake of simplicity, unnecessary whitespaces are removed from the EBNF definition. The ASE DSL consists of the specification of five elements: `Events (event)`, `Attributes (attribute)`, `ValueTypes (valueType)`, `Scales (scaleType)` and `Units (unitType)`. Events are specified as extensions to the `BaseEvent` or any already defined event. The extension uses either a single attribute or another event, as visible in Line 9. Attributes are tuples combining the *Attribute ID*, *Value Type*, *Scale Type* and *UnitType* (Line 1–3). *ValueTypes* combine a data type declaration using a primitive type and a matrix size specification of rows and columns. Additionally, the usage of the uncertainty representation needs to be specified (Line 4). The *ScalType* is limited to linear scales in the current prototype. The *ScaleTypes* are specified using three integers specifying the used ratio and the reference id (Line 5). *Units* are special, as they are defined using template expression operations to combine them. This allows an easy combination of the basic SI units to complex units according to Lines 6-10.

Listing 5.1 Generic Production Rules for EBNF of ASE specification and *Filter Expression* DSLs

```
1  cap     = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
2            "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
3            "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z";
4  letter  = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
5            "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
6            "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z";
7  digit   = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" |
8            "9";
9  alnum   = cap | letter | digit | "_";
10 name    = "_" | cap | letter , {alnum};
11 number  = digit , {digit};
12 integer = ["-"], number;
13 float   = integer , [".", number], ["e", integer];
14 bool    = "true" | "false";
```

Listing 5.2 Syntax of the ASE specification DSL in EBNF form.

```
1  attribute     = "Attribute<", name, ",", valueType | name, ",",
2                               scaleType | name, ",", unitType | name,
3                           ">";
4  valueType     = "ValueType<", var, ",", number, ",", number, ",", bool,">";
5  scaleType     = "Scale<", integer, ",", number, ",", number, ">";
6  unitType      = baseUnit | compositUnit
7  compositUnit  = "decltype(", baseUnit,"()" {"*" | "/", baseUnit,"()"}, ")";
8  baseUnit      = "Dimensionless" | "Meter" | "Second" | "Ampere" | "Mol" |
9                  "Candela" | "Radians" | "Steradians" | "Kilogram" | "Kelvin";
10 eventType     = "BaseEvent<", name, ">" | var, ["::append<", name | attr, ">"];
11 declaration   = "using ", name, "=", attr | event | valueType| scaleType |
12                                     unitType;
```

Each `Attribute` contains functions to extract the compile time information in form of an `AttributeType` to provide means to pass the TMP configuration parameters to run time. The `EventType` is created by iterating over all contained `AttributeIDs` of an *Event* and by inserting each converted `AttributeType` into a single `EventType`. The relation of `Attribute`, `AttributeType` and `MetaAttribute` follows the concept used for `Events`, `EventTypes` and `MetaEvents`. The difference is the central usage of `AttributeIDs` to identify the individual *Attributes* and relate the representations, as visible in Figure 5.3.

The mechanism available through TMP can be extended to the definition of the operations on `Attributes`' `Values` and *Filter Expressions*, which are described in the next two paragraphs.

Operations The attribute operations, as defined in Section 4.4.4, are implemented as a composition of TMP expressions and operator overloading. The TMP expressions are used to modify the *Attribute Type*. This is necessary in case of multiplication or division of `Attributes`, because the associated `Unit` needs to be changed, too. The code snippet 5.3 shows the implementation of this behavior. Line 11 computes the returned `Attribute` Type, which is the output of the operation. It uses the internal template type `Unit::mult`, which enables the multiplication of two units by the compiler, according to the definition in Section 4.4.2. This enables a direct computation and modification of `Attributes` in the CPS application without the need to convert the received ASEs before computation.

Listing 5.3 Implementation of `Unit` modification on multiplication of `Attributes`

```
1  template<typename AttributeID, typename Value, typename Unit, typename Scale>
2  class Attribute {
3    .
4    .
5    .
6    template<typename Unit2>
7    using multUnit = typename Unit::mult<Unit2>::type;
8
9    template<typename Unit2>
10   auto operator*(Attribute<ID, Value, Unit2, Scale>& b) const
11     -> typename Attribute<ID, Value, multUnit<Unit2>, Scale>::type
12   {
13     typename Attribute<ID, Value, multUnit<Unit2>, Scale> temp;
14     temp.value() = this->value() * b.value();
15     return temp;
16   }
17 };
```

The static type information is used to check the compatibility of `Attributes` as they are assigned to `Events`. In case of differences a compile time error is issued preventing run time errors. The operations on `Values` are implemented by using *Eigens Matrix* class with *ValueElement* as the basic type to support the uncertainty representation of Section 4.3.2. UML-Diagram 5.3 shows the implementation structure of the `Values` representing the *Event Data*.

Special uncertain linear algebra operations such as *norm* and *dot* are implemented through the operations of `ValueElement`. This delegation of functionality produces no overhead, because the relation between the classes is evaluated at compile time and enables full optimization by the compiler. `Value` is a *Proxy* facading the configured `Eigen::Matrix` to enable CPS applications to use the uncertain matrices like normal algebraic data types of *C++*.

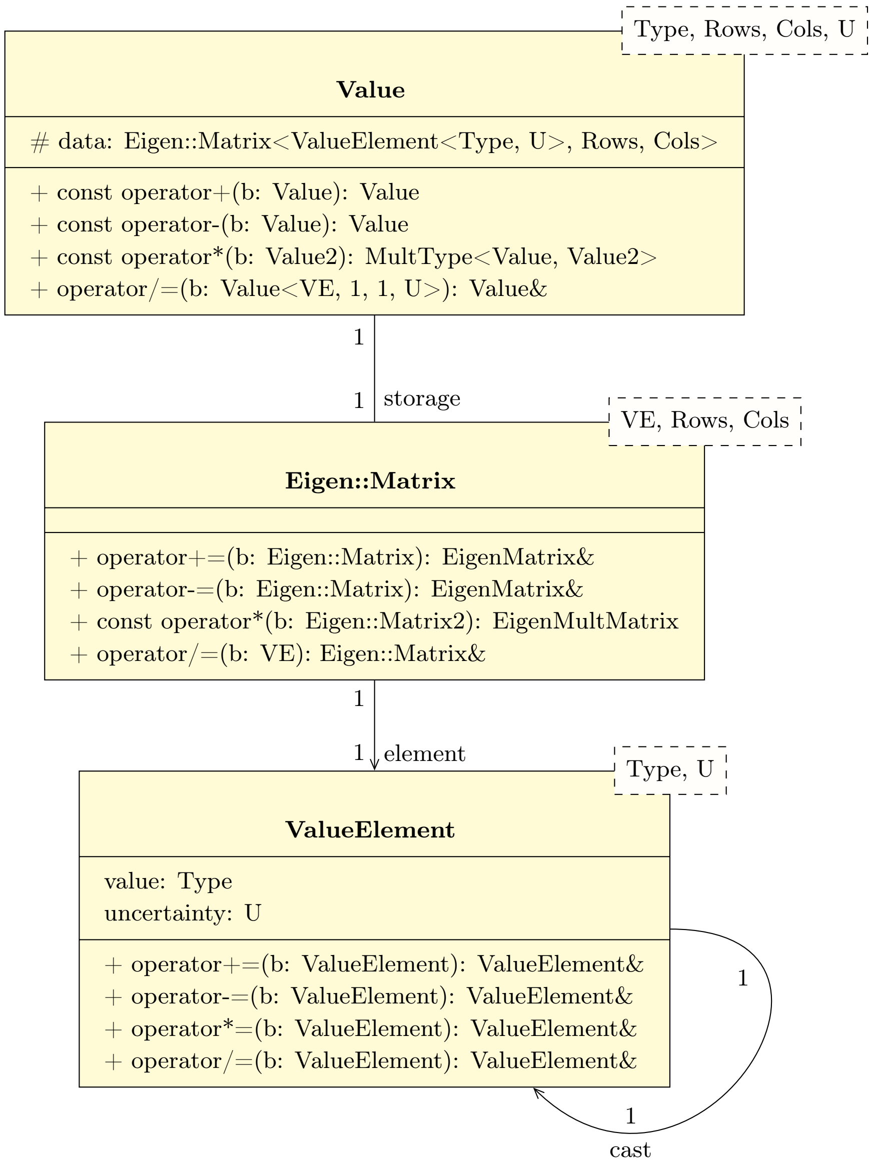

Type, Rows, Cols, U
Value
# data: Eigen::Matrix<ValueElement<Type, U>, Rows, Cols>
+ const operator+(b: Value): Value
+ const operator-(b: Value): Value
+ const operator*(b: Value2): MultType<Value, Value2>
+ operator/=(b: Value<VE, 1, 1, U>): Value&
1
1 storage
VE, Rows, Cols
Eigen::Matrix
+ operator+=(b: Eigen::Matrix): EigenMatrix&
+ operator-=(b: Eigen::Matrix): EigenMatrix&
+ const operator*(b: Eigen::Matrix2): EigenMultMatrix
+ operator/=(b: VE): Eigen::Matrix&
1
1 element
Type, U
ValueElement
value: Type
uncertainty: U
+ operator+=(b: ValueElement): ValueElement&
+ operator-=(b: ValueElement): ValueElement&
+ operator*=(b: ValueElement): ValueElement&
+ operator/=(b: ValueElement): ValueElement&
1
1
cast

Figure 5.3.: UML-Diagram showing the implementation structure of the `Values` contained in `Attributes` of ASEs in the CPS API.

CPS Filter Expressions , as described in Section 4.5.1, are necessary to state the requirements of the subscriber towards the publishers. In classical P/S systems they are used to limit the amount of information fed to the subscriber by *filtering* out irrelevant information. Additionally, ASEIA uses these expressions to configure *Homogeneous Transformations* to automatically employ sensor fusion to fulfill the requirements of the subscriber. The statement of these expressions is based on TMP, more specific on template expressions, as described by Veldhuizen in 1999 [163], to embed DSLs inside of *C++* source code investigated by Krysztof et al. in 2000 [54]. These expressions allow the formulation of lazy evaluated code. They provide the capability to specify the *static* and *dynamic Filter Predicates*, which are formulated as combinations of comparison operations, attribute assessors and boolean connectors on `Placeholders` for *Attributes* and *Values*. Some example *Filter Expressions* are shown in Listing 5.4.

Listing 5.4 Examples of `Filter` expressions.

```
1  const TimeAttr c0 = {{{ 1024 }}};
2  auto filter0 = e0[Time()] < c0;
3  auto filter1 = e0[Time()] == e1[Time()];
4  auto baseFilter0 = e0[Time()] > c0;
5  auto filter2 = baseFilter0(e0) && !baseFilter0(e1);
6  auto filter3 = (e0[Position()] != e1[Position()]) ||
7                 (e0[Time()]     >  e1[Time()]);
```

In these examples different types of *Filter Predicates* are defined. Line 1 defines a constant *Time Attribute Value*, which is used in `filter0` and `filter2` to enable *Static Selection*. The individual filters are created by using one of the predefined *Placeholders* from the *boost phoenix* [143] framework $e0, \ldots, e7$ to select the n-th input ASE. The arity of the resulting predicate is the maximum contained event placeholder. These placeholders use template expressions overloading the *C++* operators $+, -, *, /, ==, ! =, <, >, \leqslant, \geqslant$ to form complex lazy evaluation functors, which represent the specified filter. Additionally, as visible in `filter2` multiple *Static Selection* expressions can be combined using logical connectors. `filter1` and `filter3` are examples of *Dynamic Selection* as they compare attributes of different ASEs. All these `filters` are functors, which are executed on run time. Depending on the supplied object they either check the supplied ASEs or they are serialized into a network buffer to be sent to brokers or publishers. Listing 5.5 shows these two use-cases.

Line 1 and 2 define the used ASE and `PacketBuffer` used for serialization. Some example ASEs are created in the Lines 4 - 7. Lines 9 - 12 execute the filters on the example events and output text depending on the result. Lines 14 - 18 set up the environment to serialize the filters into the `PacketBuffer`. The serialization is executed in Lines 20 - 24 by supplying the special `FilterEvents` as parameters to the functors, which internally serializes the *Filter Expressions* in a network buffer. Brokers or publishers receive these and de-serialize the *Filter Expressions* in `MetaFilters` and execute them on run time. This mechanism presents a very limited mechanism to allow functions shipping within ASEIA for the purpose of exchanging *Filter Expressions*. This mechanism enables an extensive error checking on compile time, testing the existence of attributes and the validity of filter statements depending on scale, unit and other parameters of attributes. Additionally, the

Listing 5.5 Execution and serialization of `Filter` expressions.

```
using Event = BaseEvent<>;
using PacketBuffer = std::vector<uint8_t>;

Event trueEvent, falseEvent, compEvent;
trueEvent.attribute(Time()).value()  = {{{1050}}};
compEvent.attribute(Time())  = c0;
falseEvent.attribute(Time()).value() = {{{100}}};

if(!filter0(trueEvent))             cout << "False negative";
if(!filter1(falseEvent))            cout << "False positive";
if(!filter2(trueEvent, compEvent))  cout << "False negative";
if(!filter3(falseEvent, compEvent)) cout << "False positive";

PacketBuffer buffer;
auto i = std::back_inserter(buffer);
Serializer<decltype(i)> s(i);
FilterEvent<decltype(s)> s0(0, s);
FilterEvent<decltype(s)> s1(1, s);

filter0(s0);
filter1(s0);
filter2(s0, s1);
filter3(s0, s1);
```

compiler can inline and optimize the filter predicates as they are fully specified on compile time. This produces very efficient code, especially for embedded systems. The drawback of this approach is the inability to capture the *Filter Expressions* without the exact knowledge of the *Event Type*, which is necessary to execute received filters by brokers or publishers. However, the specialized *Transformation* ASE API solves this problem.

The generic *Filter Expressions* syntax is described using EBNF syntax in Listing 5.6. Lines 1–5 show the definition of matrices of value-uncertainty complexes, which are used as constants in the *Filter Expressions*. Each *Filter Expression* consists of one or more predicates concatenated using logical `AND` or `OR`, see Line 7 and 13. Each predicate consists of a comparison operation, see Line 8 applied to an attribute and another attribute or a constant, see Line 12. The attributes are defined through an event placeholder and the attribute id, see Lines 9 and 10. Each attribute can be modified by executing an unary function on it. This enables simple operations like computing the norm or separating value and uncertainty before comparing it with another attribute or constant.

5.2.3. Transformation ASE API

Transformations are considered as generic processing steps in ASEIA. Therefore, they should be specified by using minimal assumptions on the actual *Event Type*. Depending

Listing 5.6 Syntax of the *Filter Expressions* specification DSL in EBNF form.

```
value       = row, {",", row};
row         = "{", col, {",", col}, "}";
col         = "{", elem, {",", elem}, "}";
elem        = "{", float, ",", float, "}";
const       = name, " ", name, "=", value;

con         = "&&" | "||";
comp        = ">" | "<" | "<=" | ">=" | "==" | "!=";
placeholder = "e", digit;
attr        = placeholder, "[", id, "]";
attr        = name, "(", attr, ")";
pred        = attr, comp, attr | const
expr        = expr {con, expr};
```

on the *Transformations* some information may be relevant such as the *Event ID* or special *Attribute Types*. In general, the operations used in *Transformation* are specified by using *C++* constructs, that are independent from the actual implementation of the *Attribute Values* and *Attribute Scales*. This allows the designer of the system to specify *Transformations* in general and the system configures and deploys the operations as necessary. To this end, the *Object- Orientation* feature of polymorphic behavior is used. On the one hand, *Transformations* themselves are abstract classes, whose implementation is chosen on run time depending on the created *Composite Transformation*. On the other hand, the operations stated in the *Transformations* are also specified by using abstract classes encapsulating the actual data and uncertainty types. The two class hierarchies of Figure 5.4 and 5.5 show this setup for the ASEIA implementation.

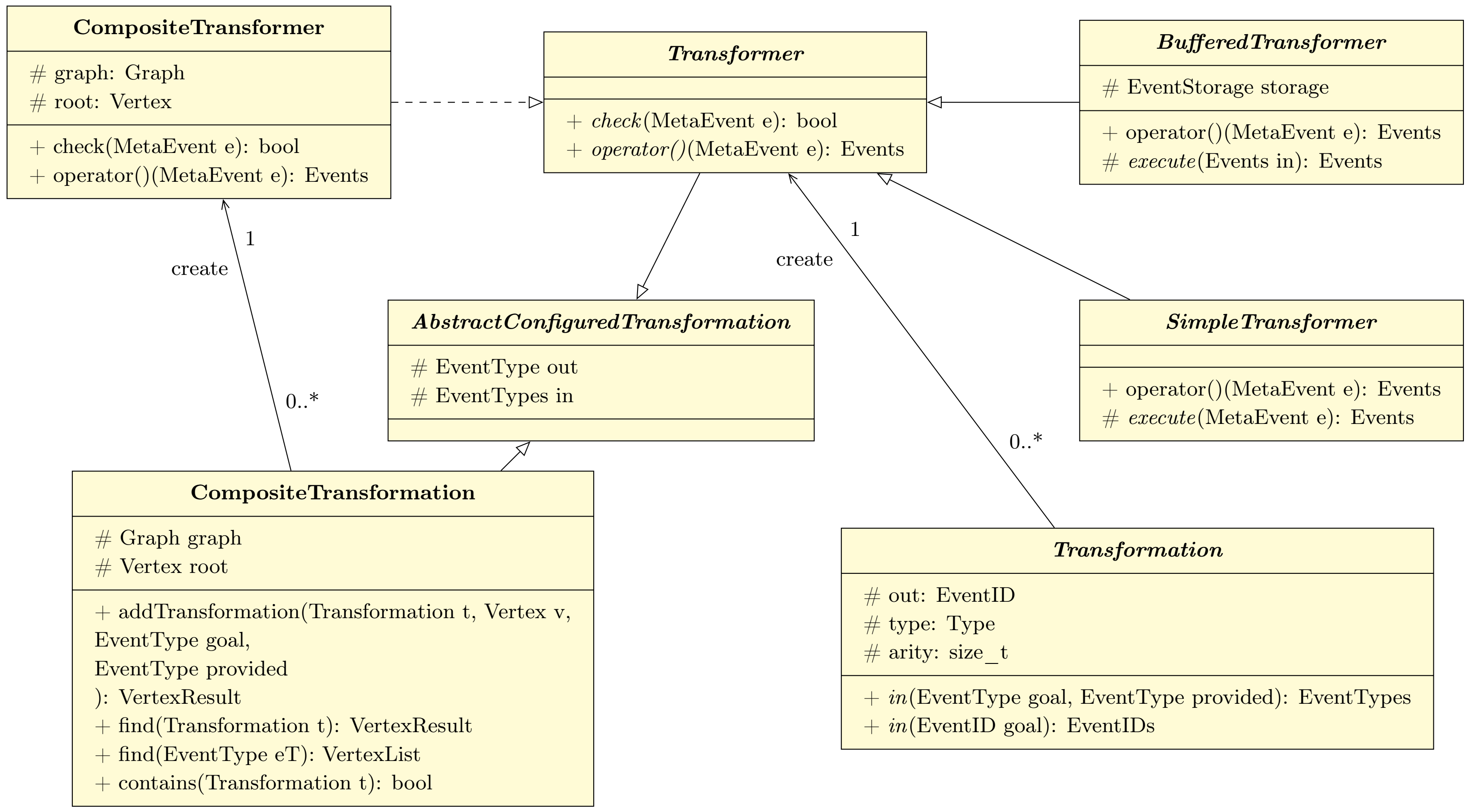

CompositeTransformer
# graph: Graph
# root: Vertex
+ check(MetaEvent e): bool
+ operator()(MetaEvent e): Events
Transformer
+ check(MetaEvent e): bool
+ operator()(MetaEvent e): Events
BufferedTransformer
# EventStorage storage
+ operator()(MetaEvent e): Events
# execute(Events in): Events
1
create
0..*
1
create
0..*
AbstractConfiguredTransformation
# EventType out
# EventTypes in
SimpleTransformer
+ operator()(MetaEvent e): Events
# execute(MetaEvent e): Events
CompositeTransformation
# Graph graph
# Vertex root
+ addTransformation(Transformation t, Vertex v, EventType goal, EventType provided): VertexResult
+ find(Transformation t): VertexResult
+ find(EventType eT): VertexList
+ contains(Transformation t): bool
Transformation
# out: EventID
# type: Type
# arity: size_t
+ in(EventType goal, EventType provided): EventTypes
+ in(EventID goal): EventIDs

Figure 5.4.: UML-Class diagram showing the Transformation class hierarchy.

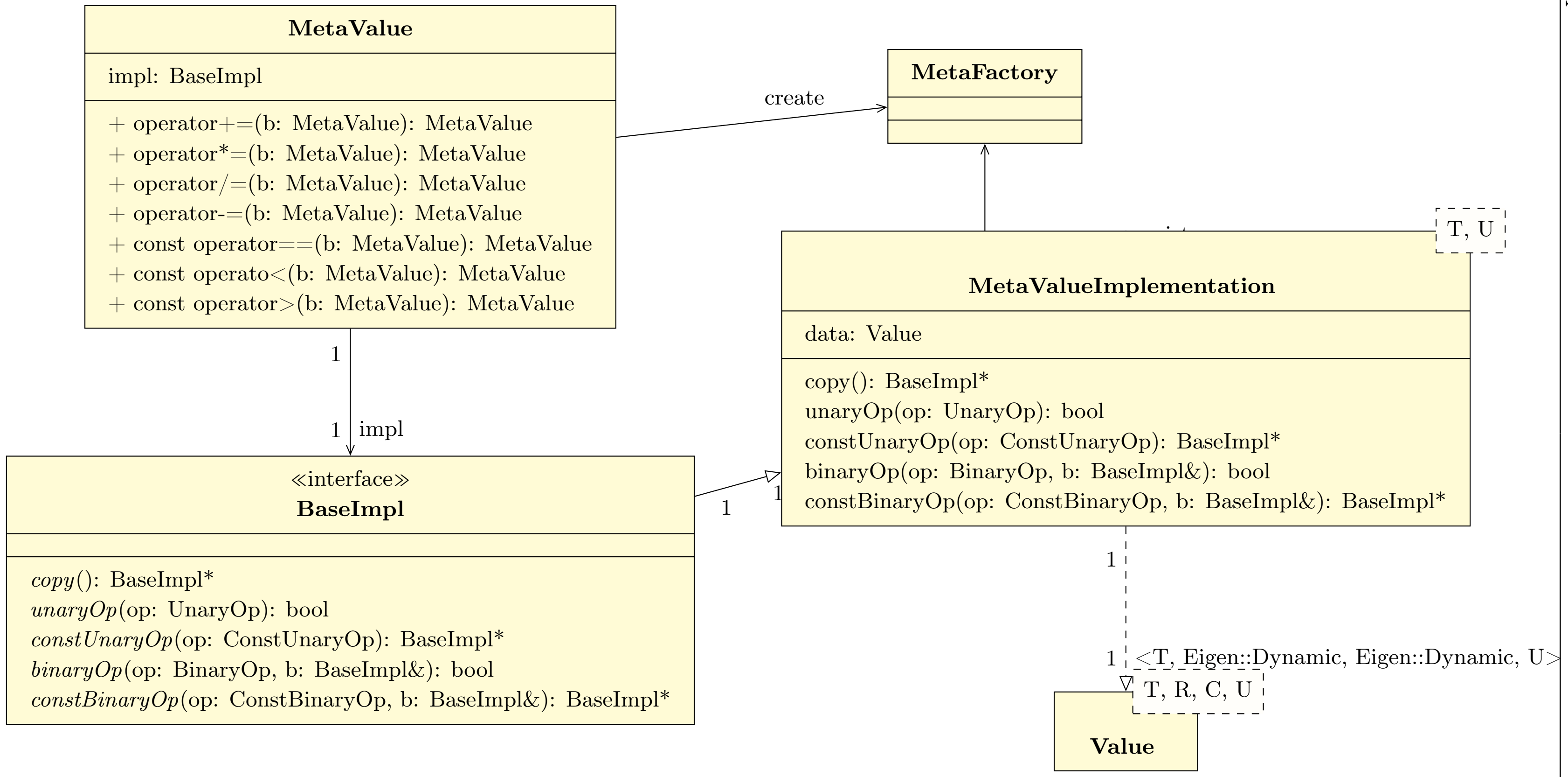

MetaValue
impl: BaseImpl
+ operator+=(b: MetaValue): MetaValue
+ operator*=(b: MetaValue): MetaValue
+ operator/=(b: MetaValue): MetaValue
+ operator-=(b: MetaValue): MetaValue
+ const operator==(b: MetaValue): MetaValue
+ const operato<(b: MetaValue): MetaValue
+ const operator>(b: MetaValue): MetaValue
create
MetaFactory
1
1 impl
«interface»
BaseImpl
copy(): BaseImpl*
unaryOp(op: UnaryOp): bool
constUnaryOp(op: ConstUnaryOp): BaseImpl*
binaryOp(op: BinaryOp, b: BaseImpl&): bool
constBinaryOp(op: ConstBinaryOp, b: BaseImpl&): BaseImpl*
1
1
T, U
MetaValueImplementation
data: Value
copy(): BaseImpl*
unaryOp(op: UnaryOp): bool
constUnaryOp(op: ConstUnaryOp): BaseImpl*
binaryOp(op: BinaryOp, b: BaseImpl&): bool
constBinaryOp(op: ConstBinaryOp, b: BaseImpl&): BaseImpl*
1
1
<T, Eigen::Dynamic, Eigen::Dynamic, U>
T, R, C, U
Value

Figure 5.5.: UML-Class diagram showing the implementation structure of MetaValues.

The `Transformation` abstract class specifies the generic *Transformation* interface, as shown in Figure 5.4. *Transformation* implementations are separated into two parts. `Transformations` describe the capabilities and the signature of the *Transformations*, whereas `Transformers` execute the functionality of the *Transformation*. This mechanism enables the registration of additional *Transformations* on run time and the dynamic execution of *Transformation* in each channel independently, as `Transformers` are created by the `create` method of `Transformations` and included in the *ASEIA Channels*. The `Transformations` can be stored in the KB as constants, which guarantee immutability to prevent run time errors created by side-effects between *Channels*. The implementation of `Transformers` and `Transformations` uses the `MetaValue` and `MetaAttribute` abstract classes to specify the functionality.

The `MetaValue` class is a *Facade* wrapping the actual polymorphic implementation of the different data type uncertainty combinations to a data type which looks like a normal *C++* arithmetic class. This *Virtual Arithmetic* enables the `Transformation` implementation to state the operations according to the interface of `MetaValues` and `MetaAttributes`. The current implementation of value uncertainty complexes in `Values` is used to implement the `MetaValueImplementations`. These are instantiated with all combinations of value types and uncertainty types to provide the *Virtual Arithmetics* run time dynamic behavior, as shown in Figure 5.5.

Additionally, the size of the `Eigen:Matrices` need to be dynamic, which can be achieved by passing the correct configuration parameter `Eigen::Dynamic` to the `Values`. The generic problem with such an approach is the creation of instances of the `MetaValueImplementations`, as they shall be completely hidden from the *Transformation* API. To this end, the `MetaFactory` registers all currently available `MetaValueImplementation` instances and enables the creation and conversion of them. This enables a run time lookup based on the `TypeID` of the value uncertainty complex.

The `MetaValue` hierarchy can be extended, but many methods need to be defined and inserted in the system to do so. Especially, the `convert` and `create` functions are necessary to enable the `MetaFactory` to enable creation and casting of the new *Virtual Arithmetic* types.

The resulting *Transformation* API is completely TMP free enabling full run time adaption. However, the internal usage of all possible `Value` implementation configurations enables exactly equal behavior between the operations of `Value` and `MetaValue`. The difference is the focus of the CPS API on performance and error prevention, whereas the *Transformation* API enables a maximum flexibility and generality of the *Transformation* specification. Listing 5.7 shows the *Heterogeneous Complementary Transformation* of two **Position** ASEs to a **Speed** ASE expressed by using the *Transformation API*.

`MetaFilter` Expressions are the run time equivalent of the CPS API *Filter Expressions*. They are designed as containers for the serialized content of the *Filter Expressions* and enable the execution of arbitrary *Filter Expressions* on `MetaEvents`. To this end, they store an expression in conjunctive normal form of comparison operations on input events' attributes or constants. The comparisons are stored as combinations of operations id , input event number and `AttributeID` and a constant or secondary input event and `AttributeID` as operand.

This enables brokers to receive the serialized *Filter Expressions* and execute them on their internal representation of the *Event Data*. Each `MetaFilter` needs to be filled through de-serialization of received *Filter Expressions* before execution. Afterwards, it represents a functor accepting a list of *MetaEvent* pointers, which it executes the de-serialized predicates on. The result of the execution is a `boolean` value representing the fulfillment of the *Filter Expressions*. Additionally, the `MetaFilter` enables access to the comparision operations it consists of on run time to create the necessary input data for the *Generic Transformation* configuration, see Section 4.6.1 and the deployment of *Homogeneous Transformations*.

5.2.4. Implementation of the Knowledge Base

The KB of ASEIA is the central data structure storing the current information on established transformations, publishers, established channels and registered data uncertainty complexes. It exists as an individual instance in each broker and subscriber to enable them to individually establish channels and deploy transformations. It is implemented as a *Singleton*, which guarantees that in each broker and subscriber only a single instance exists and that the contained data is consistent. The individual storages of the KB are associative maps. The generic structure of a KB instance is shown by the UML class diagram in Figure 5.6.

Listing 5.7 Example Transformation for Position to Speed Conversion

```
class PosToSpeedExec : public BufferedTransformer {
  public:
    BufferedTransformer(in: EventTypes) : storage({in[0]}, Newest(10)){}
    virtual Events execute(Events in) {
      SpeedEvent e=in[0];
      e.attribute(Speed()) =
        (in[0].attribute(Position()) - in[1].attribute(Position()))/
        (in[0].attribute(Time())      - in[1].attribute(Time())    );
      e.attribute(Position())+=in[1].attribute(Position);
      e.attribute(Position())/=2;
      e.attribute(Time())+=in[1].attribute(Time);
      e.attribute(Time())/=2;
      return {e};
    }
    virtual bool check(const MetaEvent& e) const {
      if(storage.empty()) return true;
      for(MetaEvent c : storage)
        if(c.attribute(Time()) != e.attribute(Time()))
          return true;
      return false;
    }
};
class PosToSpeed : public Transformation {
  public:
    Pos2Speed() : Transformation( ((ValueType)SpeedEvent()).id(),
                                  Type::Heterogeneus, 2 ) {
    }
    virtual EventTypes in(EventType goal, EventType provided) const {
      goal.remove(Speed());
      return {goal, goal};
    }
    virtual EventIDs in(EventID goal) const {
      return goal/=EventID({Speed()});
    }
};
```

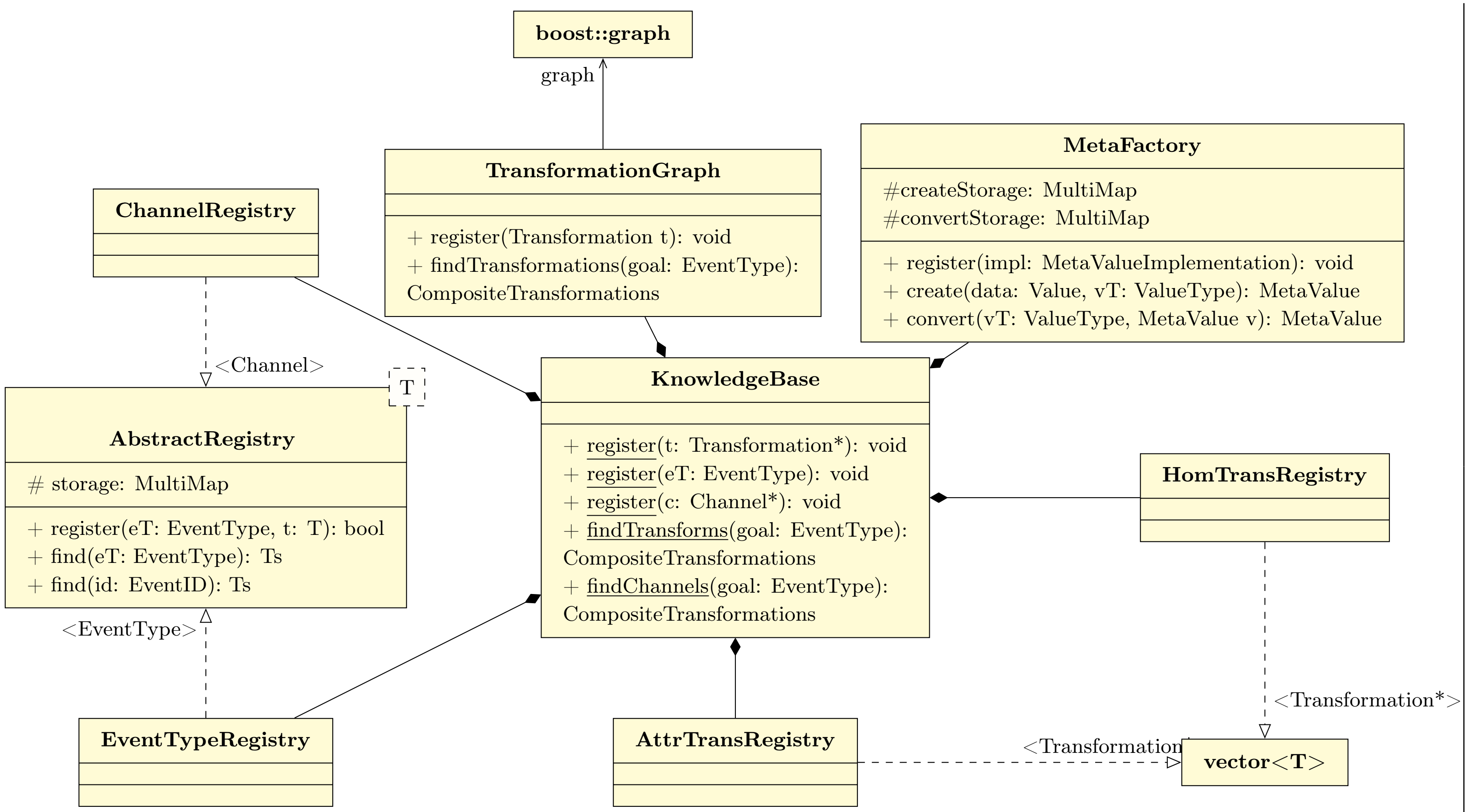

boost::graph
graph
TransformationGraph
+ register(Transformation t): void
+ findTransformations(goal: EventType): CompositeTransformations
MetaFactory
#createStorage: MultiMap
#convertStorage: MultiMap
+ register(impl: MetaValueImplementation): void
+ create(data: Value, vT: ValueType): MetaValue
+ convert(vT: ValueType, MetaValue v): MetaValue
ChannelRegistry
<Channel>
T
AbstractRegistry
# storage: MultiMap
+ register(eT: EventType, t: T): bool
+ find(eT: EventType): Ts
+ find(id: EventID): Ts
<EventType>
KnowledgeBase
+ register(t: Transformation*): void
+ register(eT: EventType): void
+ register(c: Channel*): void
+ findTransforms(goal: EventType): CompositeTransformations
+ findChannels(goal: EventType): CompositeTransformations
HomTransRegistry
<Transformation*>
EventTypeRegistry
AttrTransRegistry
<Transformation*>
vector<T>

Figure 5.6.: UML-Class diagram showing the structure of the KB Implementation. The basic data structure used is an *Associative Multi-Map* using the *EID* and the *Format Hash* as key.

Announcements are stored as pairs of *Event ID*, *Format Hash* and *Event Type*. Channels are stored as pairs of *Event ID*, *Format Hash* and *Channel*. *Homogeneous* and *Generic Transformations* are stored as lists of *Transformations* and *Heterogeneous Transformations* are stored as a graph of the *boost graph library (BGL)*[147]. *MetaeScales* and *MetaValues* are stored inside the *MetaFactory* in pairs of `id` and `create` function. The design of the KB enables a full run time update of all information as long as no id space is exhausted. The distribution of new *Transformations* is discussed in Section 5.3 and can be applied to the distribution of new data uncertainty complex implementations and new scales.

5.2.5. Implementation of Adapters

The combination of ASEIA with the underlaying P/S system is realized through adapters for the four main components of the system: *Subscribers*, *Publishers*, *Channels* and *Brokers*. The following paragraphs discuss the structure and the functionality of the adapters with ROS as the underlaying P/S system.

CPS P/S Adapters ASEIA CPS Publishers and Subscribers modify the basic behavior of the ROS Publishers and Subscribers. ROS uses *Messages* to describe the format of the exchanged data between Publishers and Subscriber. These messages are written in a special DSL, which is afterwards transformed to *C++*, *Python*, *Lisp* and *Javascript* code. ASEIA considers *Event Types* to be fully dynamic on run time, which is incompatible with ROS' static message system. Therefore, two special ROS messages are created that represent an *ASE Announcement* or *Subscription*, which is called *Channel Event* and *ASE* information itself. ASEIAs serialization system is used to fill a generic buffer contained in these messages with the run time serialized content of either information. The structure of these two ROS messages is shown in Listing 5.8.

Listing 5.8 ROS message defintion for *Channel* and ASE integration.

```
//ChannelEvent
Header header
string topic
uint64 id
uint8  type
uint8[] buffer

uint8 PUBLISHER = 1
uint8 SUBSCRIBER = 2

//AbstractSensorEvent
Header header
uint8[] buffer
```

The *Channel* information is necessary for the KB and the *Transformation Engine* to assess currently existing *ASEIA Publishers* and start the creation of *ASEIA Channels*. Therefore, it contains the identifier of the channel consisting of *Event ID* and *Format Hash* stored as a string in the `topic` field. Additionally, the type of *Channel Event* is stored in `type` as either `PUBLISHER` or `SUBSCRIBER`. The `id` field contains the unique node id of the node publishing the ROS message.

To enable CPS applications to easily access extended functionalities of ASEIA, generic `SensorEventSubscriber` and `SensorEventPublisher` classes are defined, which are configured by using an *Event Type* and for the subscriber a *Filter Expression*. These classes automatically handle periodic publications of the *Channel Events* as well as the serialization and de-serialization of ASEs. Listing 5.9 shows the creation of an example `SensorEventSubscriber` and `SensorEventPublisher` and their usage.

Listing 5.9 Code listing showing the usage of the `SensorEventSubscriber` and `SensorEventPublisher` CPS API *Adapter*.

```cpp
#include <ros/ros.h>

#include <SensorEventPublisher.h>
#include <SensorEventSubscriber.h>
#include <BaseEvent.h>
#include <IO.h>

using namespace ::id::attribute;
using ExampleEvent=BaseEvent<>;

void print(const ThisEvent& e) {
  ROS_INFO_STREAM("received: " << e);
}

int main(int argc, char** argv) {

  ros::init(argc, argv, "AseiaExample");
  SensorEventPublisher<ExampleEvent> pub;
  SensorEventSubscriber<ExampleEvent> sub(print, 10);
  ThisEvent e;

  e.attribute(Position())    = { { {0, 0} }, { {0, 0} }, { {0, 0} } };
  e.attribute(Time())        = { { {ros::Time::now().toSec(), 0} } };
  e.attribute(PublisherID()) = { { { pub.nodeId() } } };

  pub.publish(e);

  while(ros::ok())
    ros::spin();

  return 0;
}
```

Line 18 shows the creation of an ASEIA publisher of the configured `ExampleEvent`. The template class `SensorEventPublisher` is configured with the *Event Type* the CPS application wants to use. After the creation of an appropriate ASE in Line 20, the dynamic information of the event can be modified as stated in Lines 22-24. The ASE is published through a call to `publish` in Line 26. The creation is similar, but needs a callback function specified on creation of the ASEIA subscriber. This function needs to accept a immutable reference to the specified *Event Type* to ensure type safety.

Channel Adapter The *Channel Adapter* extends the basic ASEIA channel implementation, which encapsulates a *Composite Transformation*, with the capability to subscribe and publish by using the underlaying P/S system. To this end, the `Channel` provides a `handleEvent` method, which is called by the P/S specific implementation and forwards the de-serialized ASE to the internal `CompositeTransformer`. Additionally, a virtual `publishEvent` method exists that is internally called by the channel to publish all ASEs created by the `CompositeTransformer`. This method needs to be implemented by the *Channel Adapter* and serializes the ASEs into the format of the underlaying P/S system. Additionally, the *Adapter* needs to create and store the necessary information for the publishers and subscribers. In contrast to the CPS P/S Adapters no *Channel Events* are published. This mechanism allows ASEIA to provide the `Channel` class as interface and minimizes the implementation necessary to support a specific P/S system.

Broker Adapter The *Broker Adapter* consists of a special subscriber instantiated in each broker of the underlaying P/S system. This subscriber registers for *Channel Events* and feeds these to the KB. In case of *Channel Events* representing announcements, it adds the published *Event Type* to the KB's announcement list. In case of subscriptions it asks the KB to `findTransformations` fitting to the supplied *Event Type* and *Filter Expression*. The resulting list of *Transformations* are transformed into *Channels*, which are instantiated as *Channel Adapters* and stored in the KBs channel map by using the `register` method. The general broker behavior to create *ASEIA Channels* is shown in the UML sequence Diagram 5.7. This Figure shows an example sequence of events exchanged between two publishers and a subscriber. All three are considered to have different *Event Types*. After the `Broker` received the *Channel Events* containing the different *Event Types*, it creates a *CompositeTransformation* and wraps it in a *Channel Adapter*. Afterwards, ASEs transmitted from the publishers to the broker are fed to the *Transformation* through the *Channel Adapter*'s `handleEvent` method and are transformed into the subscribers goal *Event Type*. The results are delivered from the broker to the subscriber through the *Channel Adapter*'s `publishEvent` method.

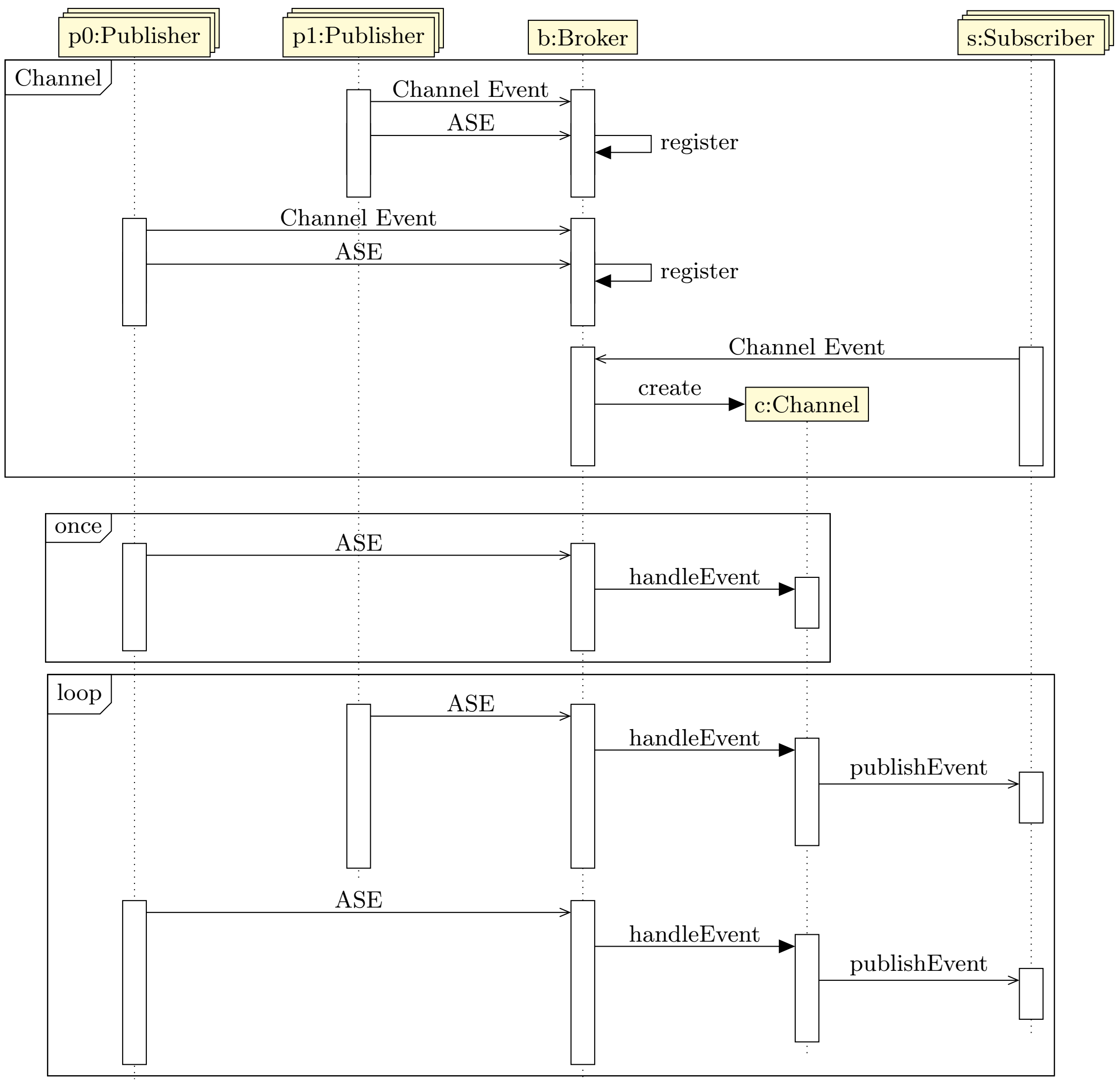


Figure 5.7.: This figure shows an example sequence of events exchanged to create and use an *ASEIA Channel*.

5.3. Extension on Run Time

As ASEIA is a generic architecture which aims at providing in-network processing for arbitrary CPS, it needs the capability to be adapted to changes in the CPS requirements and environment. To this end, multiple parts of ASEIA may be extended with new functionality on run time. The components of ASEIA that may be extended on run time are: *Attributes*, *Events*, *Data Types*, *Scales* and *Transformations*. *Events* are created by each CPS application as part of the CPS ASE API. The application specifies the basic *Event Type* and extends it with additional *AttributeTypes* or another *Event Type*. The API automatically creates the resulting *Event Type*, which can be used by the application as often as necessary and may be used directly to publish and subscribe to other CPS components. The generic CPS ASE API is described in Section 5.2.2. An example of the creation of a custom ASE *Event Type* is shown in Listing 5.10.

Listing 5.10 Listing showing an example of the creation of a custom ASE to be used by a CPS application.

```
1 #include <BaseEvent.h>
2 #include <ID.h>
3 
4 using namespace ::id::attribute;
5 
6 using Base        = BaseEvent<>;
7 using Distance    = Attribute<Distance, Value<float, 1>, Meter>;
8 using Angle       = Attribute<Angle, Value<float, 1>, Radian>;
9 using CustomEvent = Base::append<Distance>::type::append<Angle>::type;
```

The example shows the usage of the `BaseEvent` as foundation of the `CustomEvent`. Afterwards, two `Attributes` are defined for **Distance** and **Angle** information. In Line 8 the `Base` event is extended with the two `Attributes` to the resulting `CustomEvent`.

The *Transformations* use a generic ASE representation that may handle any event. Therefore, no explicit extension is necessary in this case.

The extension of existing *Event Types* with new *Attributes* mandates the creation of new *AttributeIDs*. These ids need to be unique in the whole system as they form the vocabulary of the semantic description of the ASEs' content. The introduction of new *Attribute IDs* is possible, as long as the id space is not exhausted. For a new id to be inserted in the system a consensus algorithm needs to be used, which ensures the uniqueness of the newly created id. This problem is very complex as it relates to the inference of global state in an asynchronous distributed system. Without additional assumptions the problem is not solvable as shown by Römer and Mattern [136]. The acceptable assumptions such as eventual consistency, maximum consensus time and omission degree need to be defined by the system designer and an appropriate algorithm needs to be chosen. The defintion and integration of these algorithms is considered to exceed the scope of this thesis.

Scales consist of three separate parts that may individually be extended. The type of the *Scale* is indicated by the *Scale ID*, it needs to be extended similarly to *Attribute IDs*. Additionally, a new *Scale Type* needs an implementation of the scales' scaling functions f

and f^{-1}, as described in Section 4.4.3. The mechanisms to distribute the implementations in the system are discussed in Section 5.3.2 and 5.3.3. The *Scale Reference* is indicated by a *Reference ID*, which needs to be handled similarly to *Atttribute IDs*. The third part of a *Scale* are the scaling parameters, which depend on the scaling type. These may be specified by the CPS application and are transmitted in the *Channel Events* automatically.

Data Types are extended by acquiring a new *Data Type ID* and by the distribution of the functionality necessary to compute the *Data Types* to all *ASEIA Brokers*. To this end, a new specialization of *ValueElement* is necessary, which is used by the CPS application. Additionally, the necessary functionality of `MetaValueImplementations` needs to be distributed in the system to all brokers and be registered with the `MetaFactory`. The brokers then use the registered `convert` and `create` functions to cast between the existing data types and the newly created ones. To implement new data types, the convert function needs to be defined towards all currently existing data types. The necessary instances of `convert` functions might grow very large for systems supporting many data types, as all combinations of *Value-Uncertainty complex* types need to be covered.

Extensions, which need newly created unique IDs, are expensive because of the necessary consensus in the CPS. However, these extensions typically occur seldom as *Attributes* are only created by new CPS applications not planned on design time. New *Scale Types* are seldom introduced as typical scaling mechanisms depend on the expected physical phenomena, which are known on design time. *Scale References* are often bound to nodes' or objects in the system. Therefore, these may directly use the objects or nodes' ids as reference id, which lessens the effort to create new *Scale References*. *Data Types* extensions can be considered similar to *Scale* extensions. A typical case is the introduction of a new uncertainty representation used in a dynamically added CPS application.

5.3.1. Insertion of Additional Transformations

The addition of *Transformations* on run time is the key feature of ASEIA to provide run time adaption mechanism of the CPS towards new applications and environments. To insert a new *Transformation* in the system, its parts need to be specified. In general, a *Transformation* consists of the definition of four functions and two constants. The two constants identify the arity and the type of the *Transformation*. The four functions are `in(EventID goal)`, `in(EventType goal, EventType provided)`, `check(MetaEvent e)` and `operator()(MetaEvent e)`. Additionally, the buffer needs to be configured in case the *Transformation* needs buffering. To ease the task of creating new *Transformations*, the *Transformation Hierarchy*, as described in Section 5.2.3, can be used. It defines abstract base classes that may be used to implement the *Transformation*. Depending on the type of *Transformation* different base classes may be chosen. The resulting implementation needs to be shipped to the different brokers in the system. As *C++* provides no native means of function shipping, other means need to be used to distribute the *Transformation Implementation*. Two mechanisms are supported by ASEIA: *Plugins* and *LLVM-based Interpretation*. Both approaches are discussed in the next subsections.

5.3.2. Plugins

Plugins are a mechanism to ship compiled code and integrate it into existing applications. To this end, the *Plugin* needs to adhere to a specified interface. In case of ASEIA this interface may be `Transformation` and `Transformer` to represent *Transformations* or `BaseImpl` in form of `MetaValueImplementation` to support additional *Data Types* or `MetaScale` to support new *Scale Types*. The *Plugin* is a dynamically linked library containing the necessary code to implement the specified API. It may be shipped by using any communication means to the brokers in the system and can be loaded by the dynamic linker if the system supports dynamic linking. If the system only support static linking, as typical for embedded systems, it may be integrated through a partial update of the flash, which is supported by most embedded architectures, as described by Marrón et al. [107]. The drawback of this approach is the need to know all existing architectures of brokers on run time to enable the generation of the *Plugin* for each architecture. The benefit of this approach is fast execution, as the generated code may be heavily optimized by the compiler.

5.3.3. LLVM-based compiler/interpreter

Low-Level Virtual Machine (LLVM) [96], [97] invented by Chris A. Lattner is a compiler architecture consisting of front-ends and back-ends, which use a common intermediate representation. This architecture enables LLVM to efficiently support the compilation of n programming languages on m architectures without the need to write $n \times m$ compilers. It also provides a modular optimization architecture enabling multiple optimization mechanisms to be used by the front- and back-ends. An additional feature is the run time interpretation of the intermediate representation. This enables the execution of compiled code without the need to know the target architecture of the system. It provides an easy mechanism to enable function shipping for *C++*. Consequently, brokers running the LLVM interpreter may integrate additional *Transformations Scales* and *Data Types* without the need for a specifically compiled *Plugin*. Only a generic *Plugin* needs to be provided containing the intermediate representation. However, this approach has two major drawbacks. Firstly, the intermediate code is not optimized for the target architecture, which disables architecture specific optimizations. Secondly, the LLVM interpreter needs to run on the target system, which needs at least a general purpose operating system as run time infrastructure.

LLVM might also be used to locally compile the shipped intermediate representation in the broker to mitigate the lack of architecture specific optimization. In this case only the need to provide the necessary infrastructure to execute LLVM remains.

5.4. Summary

This chapter describes the implementation of ASEIA on top of an existing P/S system. For exemplary purposes ROS has been used, but it might be exchanged by any P/S system. The chapter shows the usage of the extended functionalities of ASEIA from the perspective of a CPS application and from the perspective of *Transformations*. Additionally, it shows the mechanisms and requirements necessary to update and extend the system on run time by providing new *Transformations*, *Scales*, *Attributes*, ASEs and *Data Types*. Finally, two mechanisms to distribute and execute the run time extensions are described.

6. Evaluation

The following evaluation of ASEIA is performed to assess the usability of the mechanisms. To this end, three seperate evaluations are performed, which check ASEIA's features regarding automatic *Transformation* of events in CPS. Especially, *Time Scale Transformations* in WSN are evaluated, as these provide the foundation to many further *Transformations*. Additionally, *Heterogeneus Transformations* are evaluated in a robotic context, to enable the assessment of ASEIA's performance on real mobile entities. These *Transformations* use *Attributes* which do not relate directly to physical phenomena to check the generality of the approach. A third evaluation scenario consists of a simulation of cars in a 3D environment. This enables the assessment of the uncertainty propagation and its impact on control. The last section of the evaluation analyses the scalability of ASEIA in different network topologies and regarding different types of *Transformations*.

6.1. Hybrid Clock Synchronization

The uncertainty aware hybrid clock synchronization system is evaluated using a simulation in the *Omnet++* Network Simulator [125] version 4.2 and a small scale WSN composed of six nodes [155].

6.1.1. Simulation Setup

For the simulated evaluation this thesis uses the *INETMANET* network model [27] as well as the *MiXiM* model [90]. The implementation is distributed over two layers of the *ISO/OSI* stack. One part is located at layer 5 of the *ISO/OSI* stack and handles the transformation of time stamps for the Inter Cluster synchronization. The other is situated at layer 2 in order to gather high precision time stamps. Both layers are connected through a cross layer communication.

The evaluation focuses on the Inter Cluster Synchronization, because the simulation experiments are to investigate the scalability of the approach. The single-hop performance are evaluated by the real world experiments. Two main aspects are evaluated. The first considers the influence of the beacon period on the precision of the synchronization. This test provide information on the trade-off between message overhead and synchronization quality. The second test investigates the influence of the communication topology on the reachable multi-hop precision. It evaluates the usability of the provided time stamps for smaller and longer routes. All tests used the internal 64 bit `simtime` of *Omnet++* as reference for the synchronized clocks to evaluate the synchronization error. The `simtime` was modified by a randomly initialized drift $\rho \leqslant 10^{-5}$, to provide a realistic clock for each node. The test considered 1000 randomly created routes between nodes in the network, which were created by an optimal routing algorithm.

The simulation environment considers beacon losses, created by the collision of transmitted beacons of adjacent coordinators, and the resulting lack of information for the time synchronization. However, no data events are transmitted in the simulation. This decouples the simulations from the used MAC Algorithm and its parameters. Consequently, the simulations consider an optimal MAC-Algorithm preventing all collisions between beacons and events in the network.

Additionally, no simulation of the interrupt handling timing is done, which increases the tightness of the network. This approximation is valid, because the interrupt handling time is estimated to be smaller than $1\mu s$ because of the optimization described in Section 6.1.4.

6.1.2. Beacon Interval Analysis

The beacon interval analysis considers a rectangular grid of 50 PAN Coordinators. The area in which the nodes were distributed was $5000m$ times $5000m$. The 2.4GHz channel 11 of the 802.15.4 standard with maximum transmission power of $1mW$ was chosen. The thermal noise was fixed at $110dBm$ and the receiver's sensitivity was set to $-85dBm$. The simulation sweep started with a BO parameter of 8 until the maximum allowed value of 14. The resulting beacon interval can be computed by $BI = \frac{16 \cdot 60S \cdot 2^{\mathrm{BO}}}{SymbolRate}$. The $SymbolRate$ of the 2.4GHz band of $65.2 \cdot 10^3 \frac{S}{s}$ results in beacon intervals from $3.8s$ to $241.2s$.

Figure 6.1 shows a Box-Whisker plot of the simulation's results. The boxes represent the bounds, where 50% of all values are included. The lines represent the interval containing 75% of all values and remaining data points are included as points. As visible, with linear increasing BO values, the mean synchronization error increases exponentially. This is expected because the beacon interval also increases exponentially. Additionally, a large standard deviation independent of the hop can be observed. This is caused by the unsynchronized beacons of the individual PAN Coordinators, which might collide and therefore increase the real beacon interval. Furthermore, the data base is better for smaller hop counts, since in the given scenario short routes are much more probable than longer routes. This test proved the expected direct correlation between the beacon interval and the synchronization precision. Therefore, this value is to be considered critical for the performance of the system. Following the analyses of Section 4.5.9, the worst case performance should be better than $h2(0.0147 \cdot 2^{\mathrm{BO}}2.1 \cdot 10^{-11}s$. In all the experiments the performance was never worse than this analytical worst case bound. In the case of 1-hop long routes with a beacon order of 14, the estimated worst case precision is $9.6ms$, whereas, the experiment showed a worst case result of $1.6ms$. For beacon order 13 and a hop count of three, the worst case precision of $1.2ms$ is achieved. However, the analytically derived worst case precision in this scenario is $14.4ms$. These large differences are most likely created by random drift, which only generate the largest possible offset in rare extreme cases.

6.1.3. Topology Analysis

The second evaluation considers the performance of the system in different topologies. This is interesting, because topologies might have an influence on the length of the routes, as well as the collision probability of the beacon frames. Therefore, four basic topologies with 200 nodes each are considered. A linear (c.f. Figure 6.2a), a circular (c.f. Figure 6.2b), a grid (c.f. Figure 6.2c) and a randomly generated (c.f. Figure 6.2d) topology was chosen.

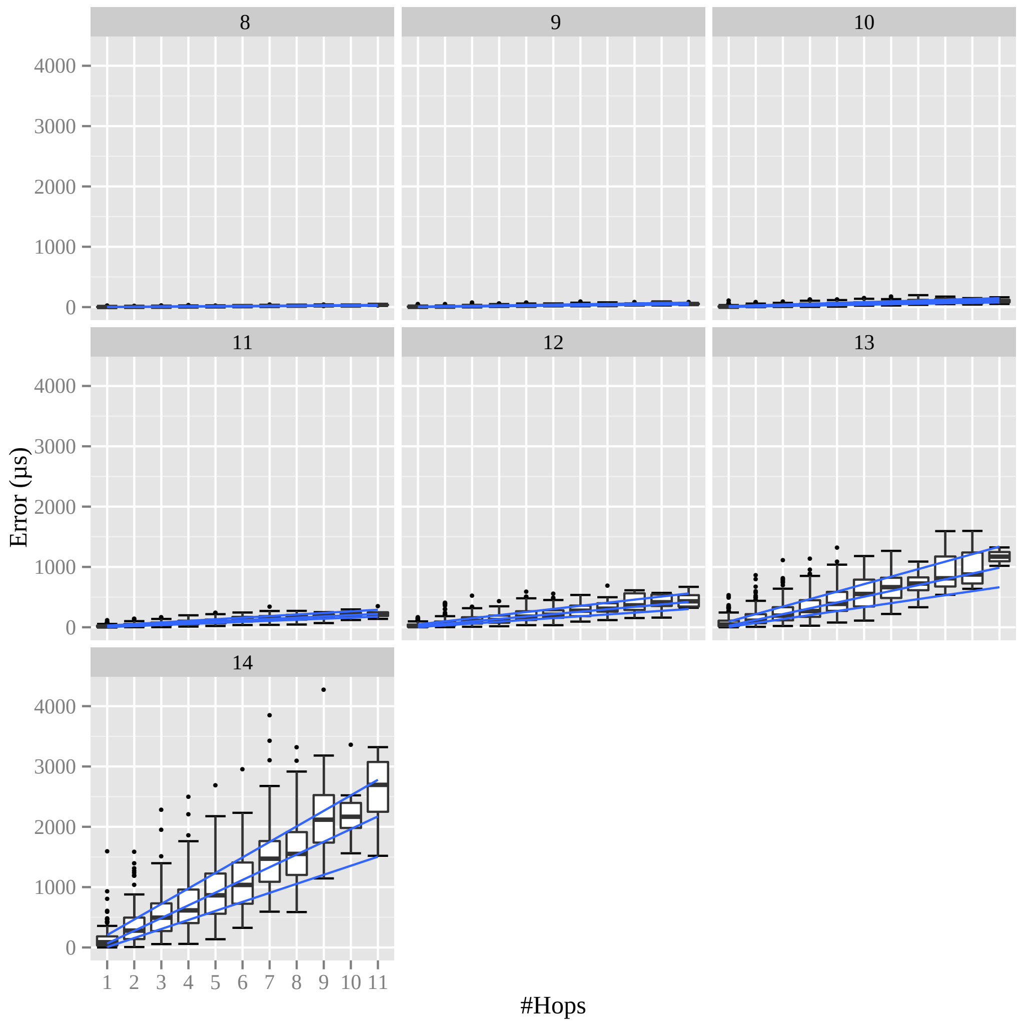


Figure 6.1.: Box-Whisker plot of the precision of a 50 node grid network with varying BO values.

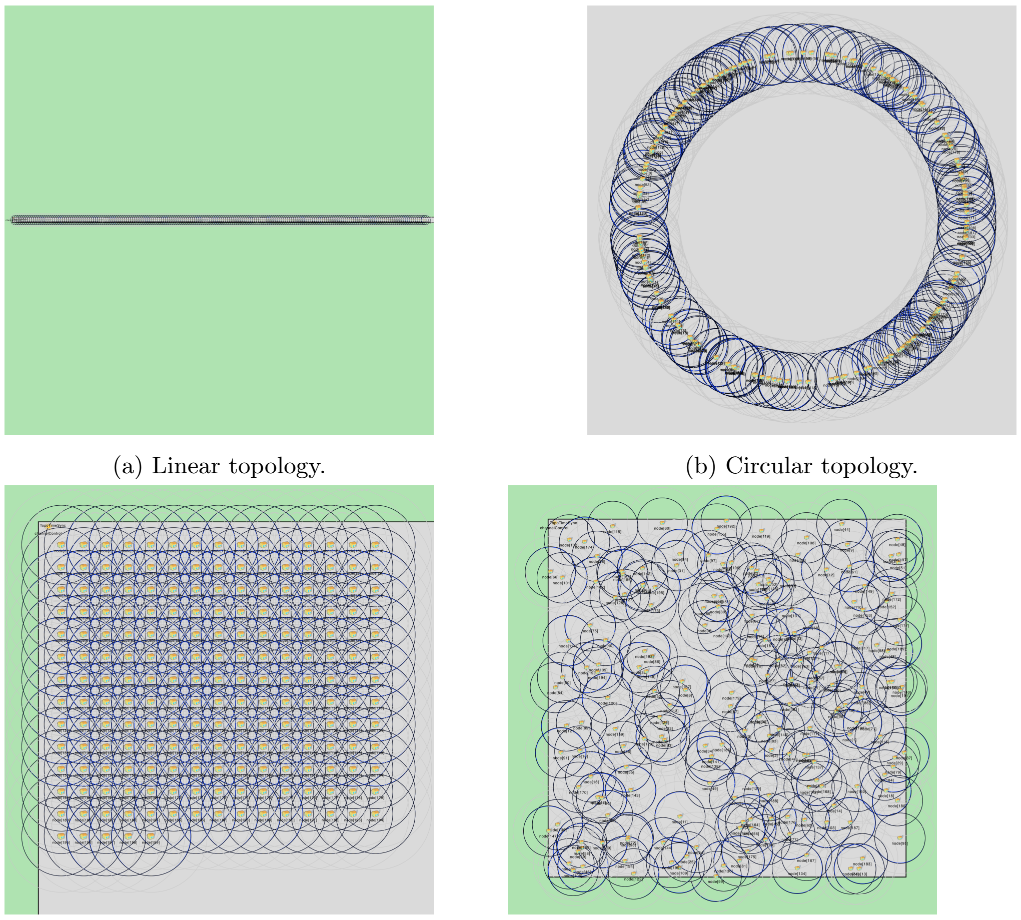

(a) Linear topology.
(b) Circular topology.
(c) Grid topology.
(d) Random topology.

Figure 6.2.: Different evaluation topologies used for the clock synchronization with 200 nodes each.

For this scenario, the same parameters as for the Beacon Interval Analysis are used, see Section 6.1.2, but with a static BO parameter of 8. As visible in Figure 6.3, the linear topology showed a maximum synchronization error of $875\mu s$. This is mainly caused by the extremely long routes created by the evaluation. The maximum hop count was 69 and for each pair of randomly selected nodes there is only one possible route. Consequently, a badly synchronized node has a large influence on the resulting precision. Considering the worst case analyses, the synchronization still performed far better than the worst case of $103.8ms$. This is caused by the very small possibility of randomly finding a long route containing 69 nodes with a maximum drift or a large amount of lost beacons. The circular topology is expected to show similar results like the linear topology, but it performed a lot better, as visible in Figure 6.4. The maximum hop count was 40 with a typical synchronization error of $150\mu s$. Compared with the linear topology, which provided a synchronization error of approximately $500\mu s$, it outperformed the linear topology. This is caused by the larger

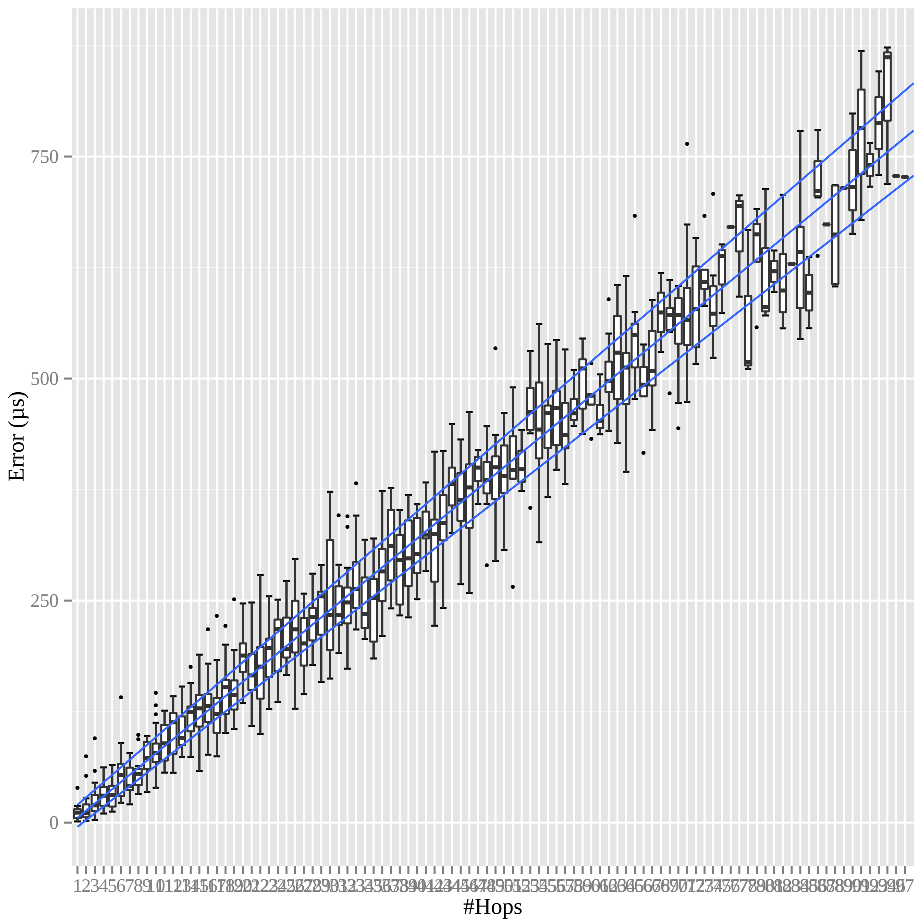


Figure 6.3.: Box-Whisker plot of the synchronization precision of a 200 Node linear topology.

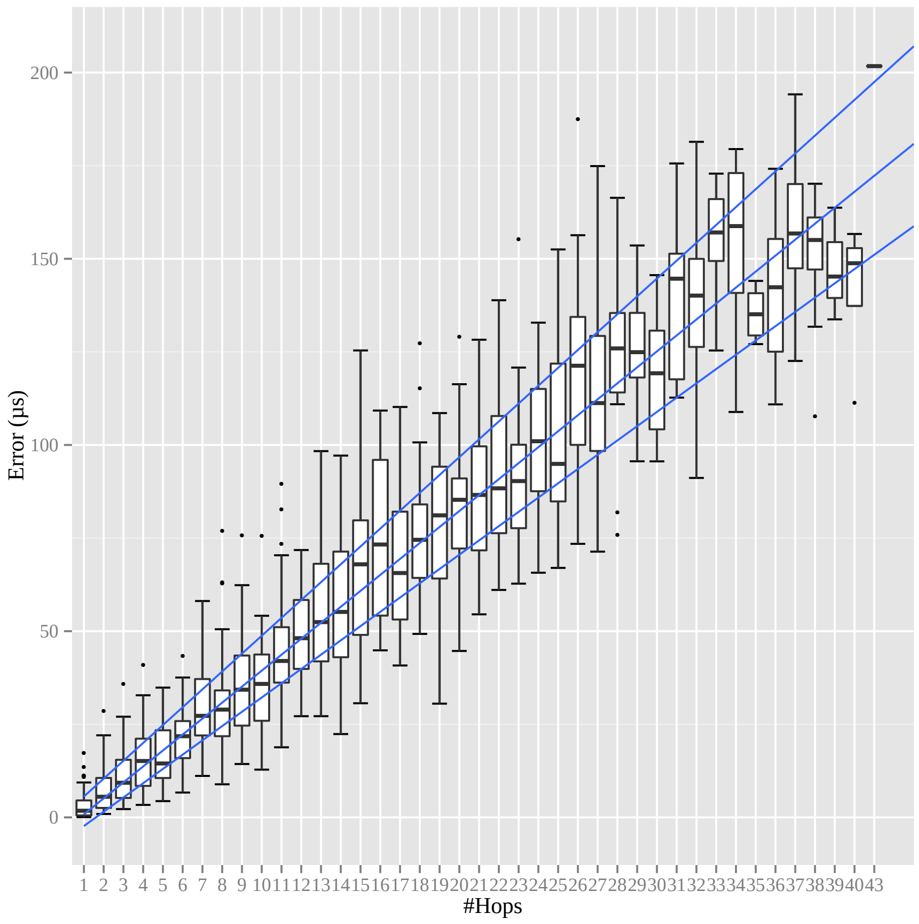


Figure 6.4.: Box-Whisker plot of the synchronization precision of a 200 Node circle topology.

amount of routes that were possible to connect two randomly selected nodes. A badly synchronized node does not have much influence anymore, since it is not as likely a part of the random route. Additionally, collisions of beacons are unlikely, because only a small number of nodes are in the vicinity of each other. In this experiment, one route of 14 hops had a comparably large synchronization error of $175\mu s$ which was still smaller than the worst case approximation of $2107\mu s$. The grid topologies (Figure 6.5) showed slightly worse

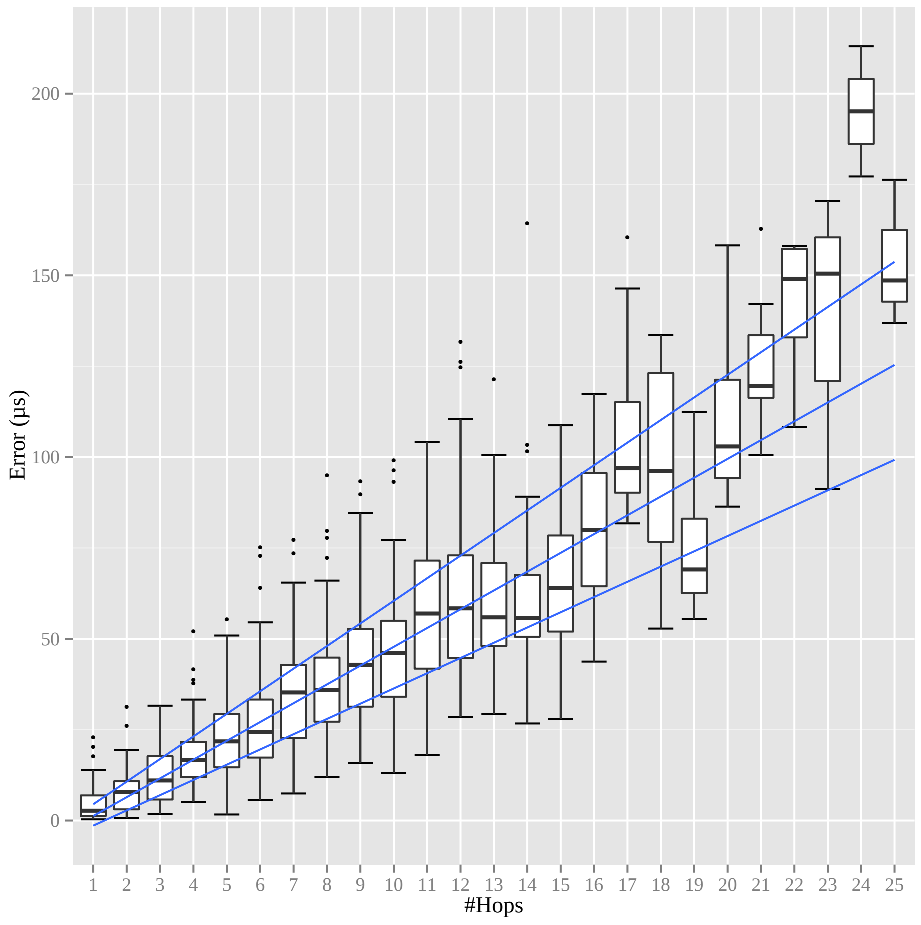


Figure 6.5.: Box-Wisker plot of the synchronization precision of a 200 Node grid topology.

performance compared to the circle topology. This is caused by the larger probability of beacon collisions caused by a higher density of nodes. At the same time, the maximum hop count is only 25, which created a maximum synchronization error of approximately $150\mu s$. The circle topology showed only a synchronization error of approximately $100\mu s$ for routes of the same length. In Figure 6.6, the performance of the random topology is visible. It shows a large deviation of individual results, which is caused by individual collisions of beacons. This problem is very dependent on the local setup of nodes around a PAN. Therefore, it is very hard to estimate and may only be observed on run time by an uncertainty evaluation. Such hot spots are likely to be contained in a randomly generated route, since hot spots contain

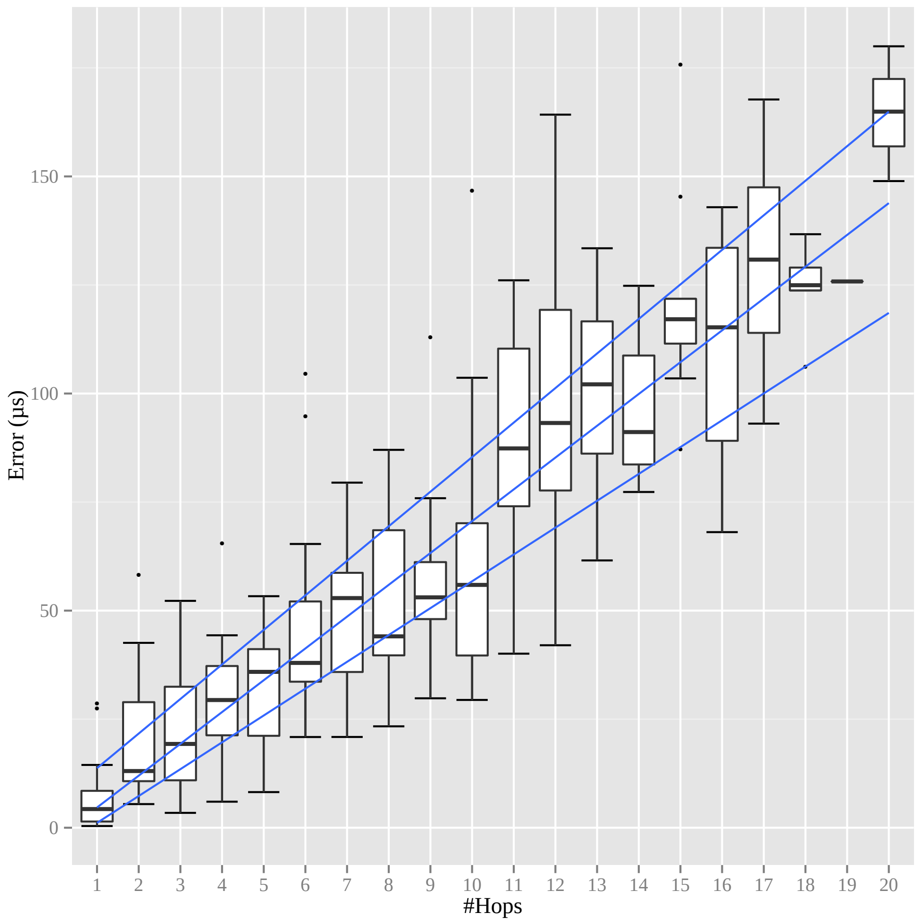


Figure 6.6.: Box-Wisker plot of the synchronization precision of a 200 Node random topology.

more nodes compared to the surrounding areas. These facts decrease the achievable mean precision. In this experiment a route of length 12 showed a large error of 161μs compared to the main quantile of the 12 hop long routes. The worst case analyses predicted a maximum error of 1806μs. Table 6.1 shows an overview on the results of the evaluation of the different

#Hop Count	Topology	Mean	Standard Deviation
1	Random	7.069008	8.495
1	Grid	5.067219	5.454
1	Linear	12.181786	9.492
1	Circle	3.327730	3.993
6	Random	47.401797	23.945
6	Grid	26.574773	13.489
6	Linear	55.861373	31.209
6	Circle	21.579193	8.327
11	Random	89.191069	27.941
11	Grid	57.463772	19.879
11	Linear	91.408097	23.836
11	Circle	45.693382	15.802
16	Random	110.255113	29.990
16	Grid	80.882388	20.756
16	Linear	131.582874	31.220
16	Circle	73.628941	21.878

Table 6.1.: Synchronization error of the simulation of different topologies in μs.

topologies. This table shows that the performance of the individual topologies towards the mean for each hop is ±50%. However, the results for the standard deviation of the tests for equally long routes show a larger difference between the individual topologies. Additionally, the value of the standard deviation for all single-hop routes is approximately the same as the mean error. This clearly indicates a need for a run time uncertainty estimation, as described in Section 4.5.9. All topologies support the mathematical analyses of the worst case synchronization error since no experiment exceeds the worst case.

As expected, in all simulations the linear topologies have the largest mean error, which is caused by the highest collision probability of the beacons. The random topology performed the second worst, which is caused by local hot spots in the topology with a lot of nodes increasing the probability of beacon losses. All topologies clearly showed a linear relation between the mean error and the hop count. This is to be expected, since the synchronization error in the vicinity of each PAN is statistically the same within each topology. The summation of the uncertainties matches very well with the increasing error in the simulation. The beacon loss probability is network specific and does not only depend on the topology, but also on the density of nodes. This probability together with the mean length of routes in the network is the major influence towards the synchronization precision.

These experiments indicate the validity of the basic assumptions. Additionally, it can observed that regular non-linear topologies provide better synchronization results. The next step is the evaluation of the performance of the approach on real hardware.

6.1.4. Small Scale Wireless Sensor Network Setup

To evaluate the correctness of the assumptions in the simulation a small, wireless network of 6 nodes is chosen to compare the results with the simulation and related work. The test network is composed of three PAN Coordinators, two Slaves and a Raspberry Pi Model A [129]. The nodes are *Cortex-M3* based devices from dresden elektronik [58] using a 2.4 GHz 802.14.5 transceiver of Atmel [30]. They run with an internal crystal oscillator and a PLL providing a clock speed of 32 MHz, which is divided by 32 to provide an internal clock with a granularity $g = 1\mu s$. The used $18.432 MHz$ oscillator has a drift of $\rho = 3 \cdot 10^{-5}$. The synchronization was implemented using hardware timers of the *Cortex-M3* microcontroller taking a time stamp on each interrupt in hardware. Using this approach, the interrupt delay between reception of the packet and the generation of the time stamp should be decreased to below $1\mu s$. The propagation time of the wireless signal in the typical range of the AT86RF232 is $t \leqslant 34ns$. The time between the reception of a packet and the generation of the interrupt by the transceiver is given as $t_{irq} = 9\mu s$. The interrupt latency of the transceiver is the same for sender and receiver and, according to data sheet, constant. Therefore, it may be omitted. The resulting tightness of the beacon network can be assumed to be $\tau \approx 1\mu s$.

The *Raspberry Pi* uses a *Preempt-RT* patched Linux kernel [69] with a real-time enabled listening program. The coordinators and the slaves use the *Atmel MAC* stack [29] to handle time stamp generation, association and beacon transmission. Both slaves and the three coordinators are connected to the Raspberry Pi through a GPIO cable as visible in Figure 6.7. Additionally, each device is connected to an evaluation PC to log the time stamps. Each coordinator establishes its own PAN, but also receives the neighboring PAN's beacons. Whenever a PAN Coordinator transmits its beacon, it also logs its internal time to the evaluation PC and toggles its GPIO. On reception of a beacon, each node transmits its virtual clock time stamp following the sender to the evaluation PC and toggles the associated GPIO-Pin. The Raspberry Pi continuously monitors the GPIO-Pins and takes a time stamp on each change. The resulting pair of *Raspberry Pi* and *Cortex-M3* time stamps are correlated and analyzed to provide an accurate offset estimation of the virtual clocks against the internal clocks of the nodes. Since beacons are transmitted unsynchronized on the different coordinators, linear interpolation to compute time stamps in between measured values is used.

The benefit of this setup is the minimal critical path, which is established by the GPIO connection. The toggling of the pin on the device is instantaneous. The available low-level access library of the Raspberry Pi provides extremely small latencies accessing the pins. Together with the used real-time program, the measurement error should be smaller than $\approx 1\mu s$.

6.1.5. Single-Hop Synchronization

This setup evaluated the single-hop synchronization mechanism. It is used as a baseline to verify the correctness of the parameters of the simulation. Therefore, the results should be close to the one hop results of the simulation. All coordinators were running and transmitting beacons, but only the two slaves ran the virtual clock towards their PAN Coordinator. The experiment is run for 5 minutes with a beacon interval of $7.5s$ and $15s$ representing a BO value of 9 and 10, respectively. The mean precision of the internal synchronization with a

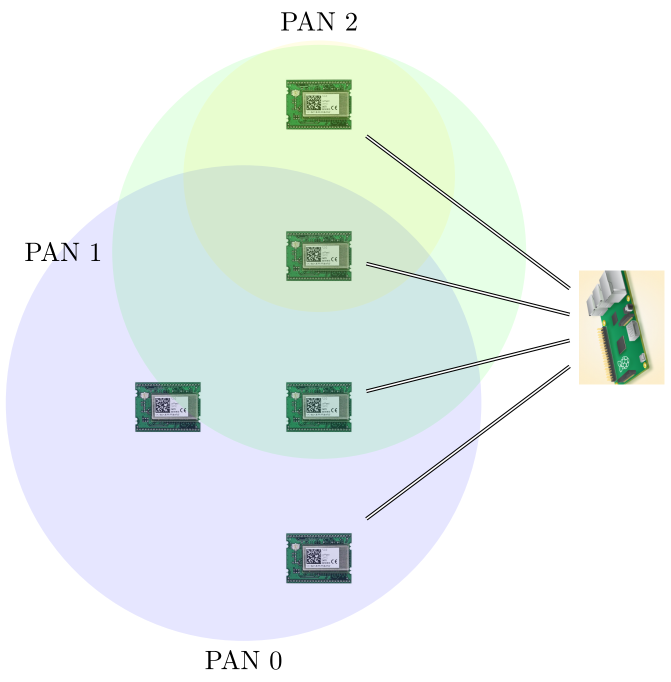


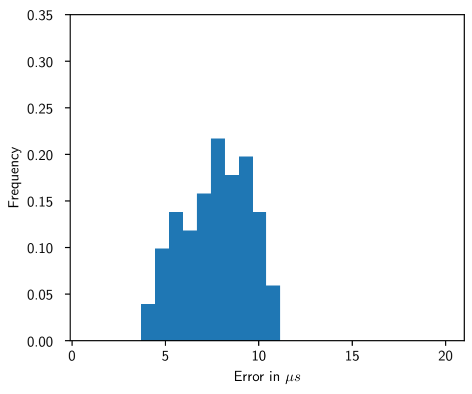
Figure 6.7.: Small scale Wireless Sensor Network composed of dresden elektronik nodes [58] and a Raspberry Pi [129] used to evaluate the hybrid synchronization.

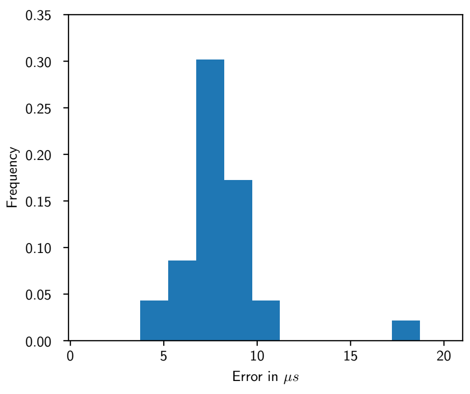
beacon interval of $7.5s$, as visible in Figure 6.8a, was approximately $8\mu s$, which fits very well to the simulated results. The deviation was approximately $\pm 1\mu s$ with the maximum error being $13\mu s$. This is less than the deviation measured in the simulation. An explanation is the smaller network size creating less beacon loss. The distribution of the values is approximately Gaussian, fitting to the assumption used in the uncertainty estimation.

Figure 6.8b shows the results of the experiment with a beacon interval of $15s$. As visible, the mean precision was also $8\mu s$ like in the previous experiment. The deviation of the precision is higher being approximately $2\mu s$ with the maximum being $20\mu s$. This is again better than the simulation results, possibly because of less beacon loss. Additionally, the crystal oscillator may be better than described in the data sheet causing less drift between the nodes.

6.1.6. Multi-Hop Synchronization

The evaluation of the Inter Cluster Synchronization used only the three coordinators. All coordinators were broadcasting beacons in this setup and ran a virtual clock for each neighboring node. The virtual clock values of the different nodes and the time stamps of the internal clocks are periodically evaluated. The used interval is the same as the beacon interval. No real event was routed through the network, since the used MAC stack provides no means of multi-hop communication. Therefore, the sum of the estimated offsets of the virtual clocks of the two pairs of nodes are compared to the offsets of the internal clocks of the first and the last coordinator. This scenario used a fixed beacon interval of $7.5s$.


(a) Precision of the *Intra Cluster Synchronization* with a beacon interval of 7.5s.


(b) Precision of the *Intra Cluster Synchronization* with a beacon interval of 15s.

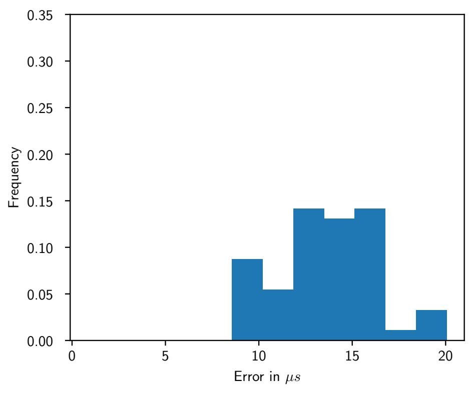

(c) Precision of the *Inter Cluster Synchronization* with a beacon interval of 7.5s.

Figure 6.8.: Histograms of the achieved precision of UHCS with different beacon intervals.

As visible in Figure 6.8c, the mean precision was $16\mu s$ with a large deviation of $\pm 5\mu s$ and a maximum offset of $20\mu s$. This result fits very well with the expected simulation results. The error is approximately doubled compared to the single-hop scenario. Additionally, the deviation has increased by the same margin.

The results of the small scale wireless sensor network experiments fully support the simulation. The baseline performance fitted very well with the exception of the larger beacon interval, which strangely showed nearly the same results.

6.1.7. Comparison with related protocols

Since the environments of the different described protocols differ, the approach is compared to available multi-hop synchronization data. *DMTS* provided a mean synchronization error of $32\mu s$ for one hop and $46\mu s$ for two hop communication. The approach performed better for single-hop ($8\mu s$) and two-hop synchronization ($16\mu s$). In comparison to *DMTS*, the approach does not need a possibly incorrect model of the latency induced by the interrupt handling of the nodes. Additionally, the granularity of the internal clocks of the nodes was far better ($g = 1\mu s$) than the ones used in the experiments evaluating *DMTS* ($g = 32\mu s$). Therefore, the results are not directly comparable. *TSAN* showed a mean synchronization error of $200\mu s$ for one hop and $1113\mu s$ for six hop communication. The approach performed better in both cases (on average $8\mu s$ and $25-37\mu s$ worst case $27\mu s$ and $110\mu s$). However, Römer et al. considered an unstructured abstract network, whereas, this approach exploited the structure and the hardware of the network to increase the synchronization precision without message overhead. Especially, the periodicity of the beacons enabled continuous synchronization, which was not used in Römer's system. Mock et al. showed a single-hop synchronization of approximately $150\mu s$, which was mainly caused by the driver abstraction and the interrupt handling of the used operating system, since they considered an experimentally derived tightness of the network $\tau = 46\mu s$. The performance of the synchronization stems from the bare-metal implementation and the good local clock increasing the networks tightness to $\tau \approx 1\mu s$.

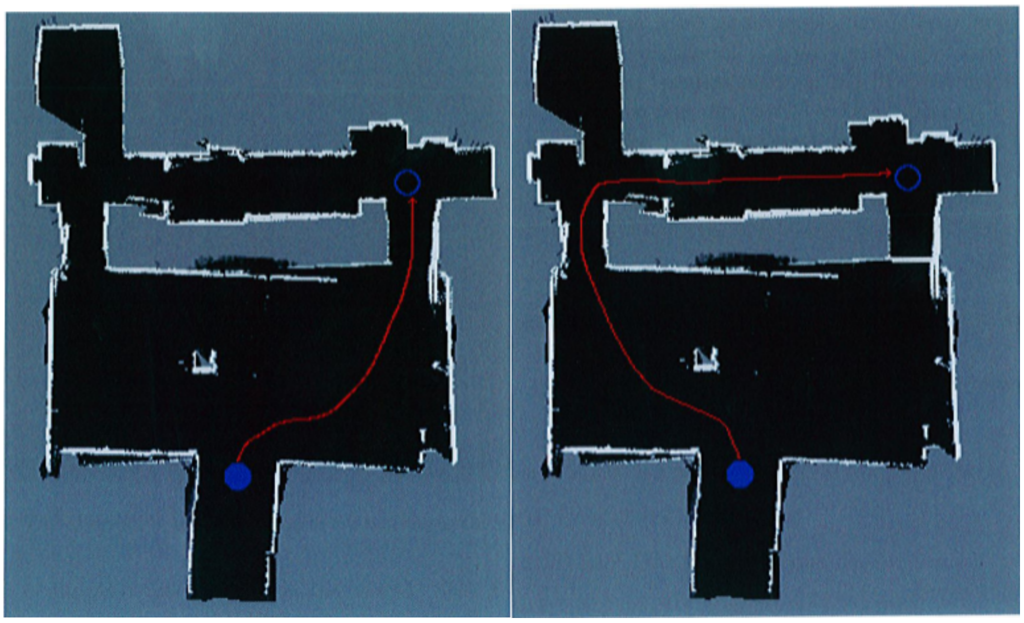
6.2. Robotic Navigation Test

The following evaluation of ASEIA's functionality was done as part of Dirk Steindorf's Master Thesis [151]. The evaluation considers the scenario of robotic navigation in dynamic instrumented environments. The goal of this evaluation is the demonstration of ASEIA's capabilities in a real world setup. Additionally, the evaluation shows the integration of ASEIA into an existing CPS setup using ROS.

6.2.1. Scenario

The goal of this evaluation is the testing of ASEIA's capabilities to process complex structured information and its context in a real scenario. The scenario consists of a *MetraLabs Scitos G5* robot driving autonomously in the lecture hall of the Faculty of Computer Science of Otto-von-Guericke University Magdeburg. The lecture hall consists of a single room with tables and chairs and three doors. Two of these doors lead to a hallway on the north side of the lecture hall, whereas, the third door leads to a second hallway on the south side of the hall. The evaluation considers different scenarios regarding the state of the doors on the north side. As the faculty did not allow changes to the lecture hall itself, virtual rotation encoders are used, which are represented by laptops running ASEIA publishers. These publish the current door opening angle through manual operation. Additionally, each publication contains information on the context consisting of door position, sensor identification and time. The robot's goal was to drive from the middle of the lecture hall to the north west corner of the northern hallway. Figure 6.9a shows the layout of the lecture hall with the two doors and the optimal path. The problem to solve is the integration of the doors as dynamic obstacles, because the state of the doors has a large influence on the chosen path of the robot.

To this end, existing navigation software is used, which is fed with dynamically generated occupancy grid maps. The maps consist of a static part, which are extracted from the



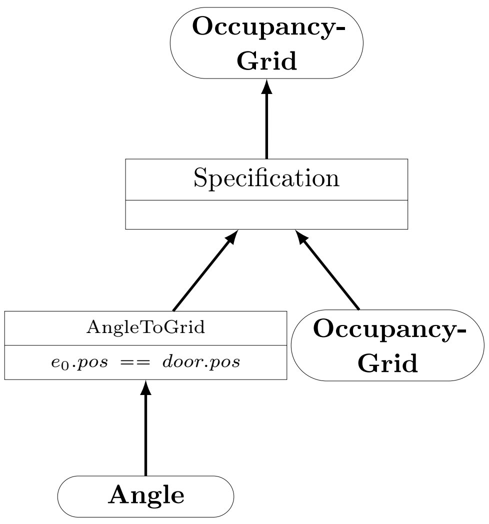
(a) Visualization of the the optimal shortest path which is assumed by the robot before any environmental information is included or when both doors are open.

(b) The optimal path in case the right door is closed. The fusion of the occupancy grids can be seen by the added obstacle at the right door.

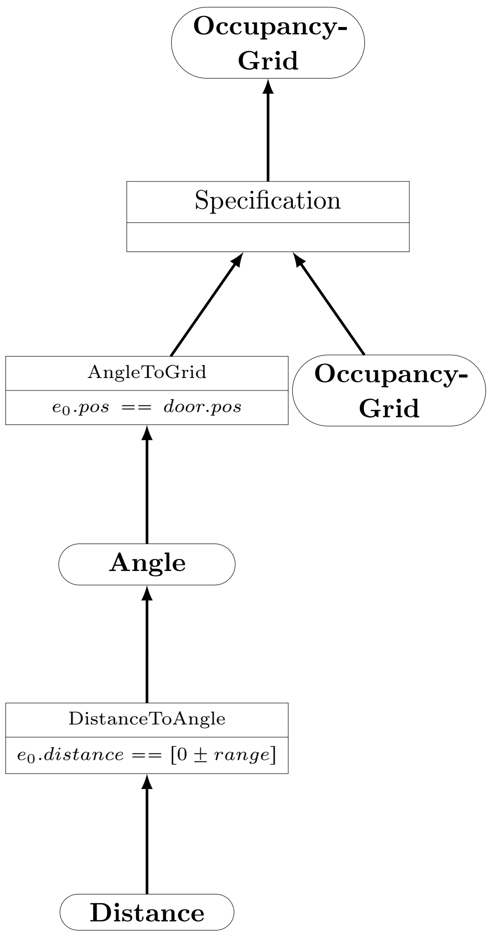
Figure 6.9.: Visualization of the occupancy grid maps of the lecture hall of the Faculty of Computer Science of the Otto-von-Guericke University Magdeburg. Additionally, the start point and the goal of the robot are visualized as a filled and a hollow blue dot. The red path indicates the planned path of the robot.

mapping step of the navigation software. Additionally, the rotational encoder information is transformed using a *Heterogeneous Unary Transformation*, see Section 4.5.4 to a **Door Event**. Afterwards, the **Door Event** is transformed to an **OccupancyGrid** centered at the position of the door containing the door as obstacle. The resulting map contains only a small part of the environment. Compared to the static map, the information is more certain, because the source information used to generate the dynamic map is newer. To combine both **OccupancyGrids**, the *Specification Transformation*, see Section 4.5.6, is used, which overwrites the content of the static map with the current door state. The resulting map is kept to include additional information. To overcome the problem of the map not updating because minimum uncertainty is reached, the maps had a limited lifetime defined in the subscriber of the navigation software. Therefore, old maps are automatically discarded by the system. The resulting transformation network is shown in Figure 6.10.

The very good *Mira Navigation Stack* [60] of the *Scitos G5* could not be used because it did not allow dynamic updates of the provided occupancy grid map. Consequently, the evaluation was performed using the *ROS Navigation Stack* [106]. To map ASEIA to the ROS interface of the navigation stack, a special forwarding node is implemented that translates ASEIAs **OccupancyGrid**s to *ROS maps.*



(a) **Angle**-based *Transformation Graph.*



(b) **Distance**-based *Transformation Graph.*

Figure 6.10.: Graph showing the resulting *Composite Transformation* used to enhance the static occupancy grid of the robotic test scenario with dynamically generated rotation information representing the door states of the lecture hall. The resulting dynamic occupancy grid map has the same size and resolution as the static one, but reduced uncertainty.

Listing 6.1 Source code of the **Angle** publisher for the robotic evaluation scenario.

```
 1 using AngleVal  = Value<int16_t, 1, 1, true>;
 2 using AngleAttr = Attribute<Angle, AngleVal, Radian, Scale<1, 100, 0>>;
 3 using ThisEvent = BaseEvent<>::append<AngleAttr>::type;
 4 
 5 SensorEventPublisher<ThisEvent> pub;
 6 
 7 void run(const ros::TimerEvent& msg) {
 8   ThisEvent e;
 9   e.attribute(Time()).value()        = {{{std::time(nullptr), 1}}};
10   e.attribute(PublisherID()).value() = {{{55}}};
11   e.attribute(Angle()).value()       = {{{147, 10}}};
12   e.attribute(Position()).value()    = {{{232, 0}}, {{574, 0}}};
13   pub.publish(e);
14 }
15 ...
```

The usage of the CPS API, as described in Section 5.2.2, is shown in Listings 6.1 and 6.2. Listing 6.1 shows an example **Angle** publisher. The **Angle** event is specified as an ASE according to the ASE DSL in Line 1-3. The **Angle** event consists of a `BaseEvent` and an `Angle` attribute. The `BaseEvent` contains data on position, time and publisher id. This information is later used to merge the generated **OccupancyGrid**s. The **Angle** specification consists of *C++ TMP* statements, which are evaluated by the compiler on compile time. It specifies the scale, value structure, value type and unit of the sensor data and context attributes contained in the event. Line 5 declares a publisher using the specified **Angle** event. This publisher only accepts instances of this event type. This prevents programming errors, like mixing up different types of events. The only parameters of the event, which may change on run time are the values and uncertainties of the sensor data and the context attributes. On assignment their structure is checked and if necessary conversions are done (Line 9–12). If a context attribute is unset or the uncertainty is unset, a default value is used. The default value consists of a value-uncertainty compound indicating maximum uncertainty. This enables the detection of the programming error in the receiving component if the uncertainty is evaluated.

Subscribers use the same specification language as publishers regarding the ASEs. This is visible in Listing 6.2, which shows parts of the translation program converting **OccupancyGrid** ASEs to ROS *OccupancyMaps*. Similar to the publisher the necessary attributes, their data type, scale and unit are specified. This time no assignment is done, but the subscriber is supplied with a callback function, which is executed on reception of a fitting ASE. The structure of the received ASE is guaranteed by ASEIA, which enables the usage of the contained grid attribute according to the specified size. The developer does not need to care for `out_of_range` errors or segmentation violations. If the original event contains a grid of different size, it will automatically be resized. If no information is present for certain grid cells, the grid cells are initialized with the default value and maximum uncertainty. The uncertainty cannot be transferred to the ROS message, but the ASE attributes automati-

Listing 6.2 Source code of the **OccupancyGrid** subscriber for the robotic evaluation scenario. The resulting **OccpuancyGrid** is translated to the native ROS message and forwarded to the *ROS Navigation Stack*

```
1  using GridVal   = Value<uint8_t, 1024, 1024, false>;
2  using GridAttr  = Attribute<OccupancyGrid, GridVal, Dimensionless,
3                              Scale<1, 1, 0>>;
4  using GridEvent = BaseEvent<>::append<GridAttr>::type;
5
6  void forward(const GridEvent& e){
7    nav_msgs::OccupancyGrid msg;
8    msg.data.resize(1024*1024);
9    auto occ = e[OccupancyGrid()].value();
10   copy(occ.begin(), occ.end(), msg.data.begin());
11   auto x                       = e[Position()].value()(0);
12   auto y                       = e[Position()].value()(1);
13   msg.header.stamp             = e[Time()].value()(0);
14   msg.info.resolution          = 0.05f;
15   msg.info.width               = 1024;
16   msg.info.height              = 1024;
17   msg.info.origin.position.x = x.value()-x.uncertainty();
18   msg.info.origin.position.y = y.value()-y.uncertainty();
19   msg.info.origin.position.z = 0;
20   pub.publish(msg);
21 }
22
23 int main(int argc, char** argv){
24   ...
25   SensorEventSubscriber<GridEvent> sub(1, forward);
26   ...
27 }
```

cally support the extraction of the value without uncertainty. This enables the integration of ASEs and ASE attributes with existing software. Line 9 shows the usage of a *C++ STL* algorithm to transfer the contained grid data to the ROS message. Additionally, the CPS API enables the developer to use the received ASEs without any additional conversion in linear algebra expressions. Line 16 and 17 show the extraction of the position of the lower left pixel of the map to convert the position to the expectations of ROS. `value()` and `uncertainty` separate the attribute into its value and uncertainty part respectively. The benefit of the strict ASE specification is the guarantee of the system that any received event contains exactly the requested information. If the specified ASE does not fit any publisher and no *Transformation Graph* can be established, no channel is created and no data will be received by the subscriber. This enables the subscriber to execute defined failure mitigation behaviours instead of trusting false information.

6.2.2. Tests

The evaluation considers two test cases. The first test case uses the **ROS Navigation Stack** as baseline to drive the robot from the start position to the goal position. The second test uses the extended functionality implemented using ASEIA to enhance the behavior of the baseline test in case the shortest path to the goal is blocked by a closed left door. The execution of the test cases can be observed in the two videos `baseline.mp4`[1] and `extended.mp4`[2].

Baseline Navigation Stack The baseline Navigation Stacks always choose the plan of Figure 6.9a as the global plan. If the left door is open, the global planner generates the shortest path and the robot follows this path without problems. If the right door is closed the robot starts to follow the optimal path, but as soon as the laser scanner of the robot detects the closed door, the local planner of the robot tries to circumvent the obstacle. This is not possible in the scenario, which results in the global planner searching for an alternative route to the goal. This generates the alternative path through the left door. The robot now drives to the left door. If the left door is closed, the robot again tries to find an alternative path again, which fails and the robot stops. If the door is open, it follows the new path to the goal. The necessary detour to detect the closed door was rather long, because the laser scanner of the robot has only 8 m range. The time until detection of the unavailability of a path to goal in case both doors are closed is also very long and contains much unnecessary movement.

Extended ROS Navigation Stack The extension Navigation Stack showed exactly the intended behavior. In case of a closed right door the robot directly planned a route through the left door according to Figure 6.9b. In case both doors were closed, the robot did not start to move because a valid plan could not be found. This decreased the necessary movement time in case of closed door and reported errors faster than the baseline Navigation Stacks. However, the design of the Navigation Stacks did not enable an en-route detection of changes in the **Door Events**. If a door was changed while the robot was driving, the global plan remained unchanged unless an obstacle was detected, which required replanning. This problem can only be solved by a change to the used architecture of the ROS Navigation Stack, which by far exceeds the scope of this evaluation.

6.2.3. Results

The results of the evaluation show the applicability of ASEIA to dynamically integrate available sensor data to enhance robotic navigation. The *Transformation* infrastructure enables a dynamic composition of environmental information sources and robotic information sinks to robotic applications. To implement the system, only an adapter between *ROS Navigation Stack* and ASEIA and the *Heterogeneous Unary Transformation* need to be implemented as the environmental sensors are considered to exist in the environment. Finally, the static

[1] `https://github.com/steup/ASEIA-ROS/blob/sensor-events/src/aseia_examples/common/baseline.mp4`

[2] `https://github.com/steup/ASEIA-ROS/blob/sensor-events/src/aseia_examples/common/extended.mp4`

OccupancyGrid of the environment needs to be published. The actual *ROS Navigation Stack* never needs to be modified. Additional tests show the benefits of the approach by replacing the virtual rotation sensor with a virtual door sensor, which yields the same results without changing anything in the configuration of the *Transformations* or the navigation stack. The used *Transformation Graph* is automatically extended to include an additional *Transformation* as visible in Figure 6.10b.

The resource consumption is low. Table 6.2 shows some statistical numbers regarding used communication bandwidth and CPU utilization of the *Transformations*. The results were acquired with a laptop running *Ubuntu 14.04* and *ROS Indigo*. The machine was equipped with an *Intel Core i5 2520M* processor and 8 GiB of RAM. The CPU consumption was measured using *perf* [57], whereas, the network throughput was measured using *Wireshark* [118]. To enhance the statistical significance of the measurement, all experiments are performed five times with two different setups. In these cases the robot is stopped, but receives the dynamically generated occupancy grid maps and executes its path planning. Each measurement was done with a publication rate of 1 Hz and 10 Hz.

	1 Hz	10 Hz
Total Run Time	599.63 s	599.70 s
Used CPU Time	0.47 %	1.89 %
Time: **Angle Event**	9.87 %	7.02 %
Time: **Door Grid Event**	8.72 %	7.23 %
Time: **Robot Grid Event**	9.43 %	5.77 %
Traffic: **Angle Event**	380.00 $\mathrm{B\,s^{-1}}$	3800.00 $\mathrm{B\,s^{-1}}$
Traffic: **Door Grid Event**	762.00 $\mathrm{B\,s^{-1}}$	7620.00 $\mathrm{B\,s^{-1}}$
Traffic: **Robot Grid Event**	382.00 $\mathrm{B\,s^{-1}}$	3820.00 $\mathrm{B\,s^{-1}}$

Table 6.2.: Table showing statistical results of the evaluation of the robotic scenario. The *Transformations* time is expressed as a fraction of the whole CPU time used.

The results clearly indicate the performance of the *Transformations* implemented in ASEIA. Even though not exactly tested, the implemented transformation should be executable on microcontrollers, because of their limited usage of CPU and network resources.

6.3. Automotive Scenario

The automotive scenario, explained in Section 1.2.2, provides multiple physical phenomena in dynamic environments, which are well suited to test the capabilities of ASEIA regarding specification of context, quality of context and the propagation of this information through the processing. The automotive test cases are designed to be highly reproducible and enable a quantitative evaluation of ASEIA's performance. To this end, a physical simulation of car behaviors is needed in a sufficiently complex environment to produce realistic sensor output. The virtual cars should be fully autonomous, as any human control limits the amount of testing. The behavior of the cars uses the *Karyon* hybridization kernel approach to approximate a realistic safe system [52] consisting of minimum functionality, which is guaranteed to be safe, and extended behavior using external sensors as also done by Berger et al. [36]. The cars themselves combine the autonomous behaviors of *Lane Following* and *Adaptive Cruise Control* on a circular street to enable an unlimited test time. The street itself approximates a rural worst case scenario with sharp turns, hills and mountains blocking the direct perception. Finally, all sensor contain a configurable uncertainty model to enable the evaluation of the uncertainty propagation within ASEIA.

6.3.1. Scenario

The scenario uses a custom build simulation scenario implemented in *Virtual Robotik Experimentation Platform (V-Rep)* [133]. It contains a circular road in mountainous terrain with multiple cars. The cars are equipped with virtual sensors enabling the detection of their **Position**, **Orientation**, **Speed**, and their minimum **Distance** to any car in front. The cars will perform ACC and *Lane Following* autonomously using their on-board sensors. The goal of the scenario is the evaluation of an extended behavior called *Extended ACC*, which uses additional **Pose Events** to infer **Distance Events** and combine them with the local observation to enhance the behavior of the local ACC. This *Kalman-based Distributed Virtual ACC Sensor* is described and evaluated in following sections. It can be considered and extension of the *Collision Warning System*, described in Section 1.2.2. The scenario enables a coverage of ASEIA's functionalities regarding *Event Description*, *Event Dissemination*, *Event Filtering* and *Event Fusion*. Additionally, the quality of the resulting information can be evaluated using the propagated uncertainty of the information and the behavior of the simulated cars.

6.3.2. Implementation

The evaluation simulation is implemented using *ROS* as communication middleware controlling and interacting with the *V-Rep* simulation. ASEIA provides, on top of *ROS*, semantic annotation, context description and transformation deduction and execution. The autonomous behavior of the cars is implemented directly as *ROS* nodes. The whole system is running in lock-step mode to enable full reproducibility of the results. The implementation can be found in the *ASEIA-ROS GitHub* repository [153].

The generic architecture used to control the cars and generate the virtual sensor information from *V-Rep* is shown in Figure 6.11.

As visible, the central node `sim` controls the *V-Rep* simulation and enables the lock-step updates of the `car` nodes. The `car` nodes contain virtual sensors, controllers and actuators. The virtual sensors and actuators fetch and deliver their data directly to and from the *V-Rep* simulation through *V-Rep's ROS API*. This API is service-based and enforces synchronization between *V-Rep* and the `car` nodes. The `car` nodes provide their virtual sensor information as *ASEIA Publishers* and subscribe for enhanced sensor information using *ASEIA Subscribers*. The *manager* node contains a single *ASEIA Broker* providing the ability to create *Channels*, deduce *Transformation Graphs* and execute *Transformations*. The `record` node is used to gather the *Events* during a simulation run for later analysis.

The used virtual ACC sensor implemented in the cars outputs the distance to the next car in front or the maximum distance. The `car` nodes fetch their virtual sensor information directly from the *V-Rep* simulation and add a configurable Gaussian-noise to the gathered data. The noise is produced through the execution of a *Mersenne-Twister* with

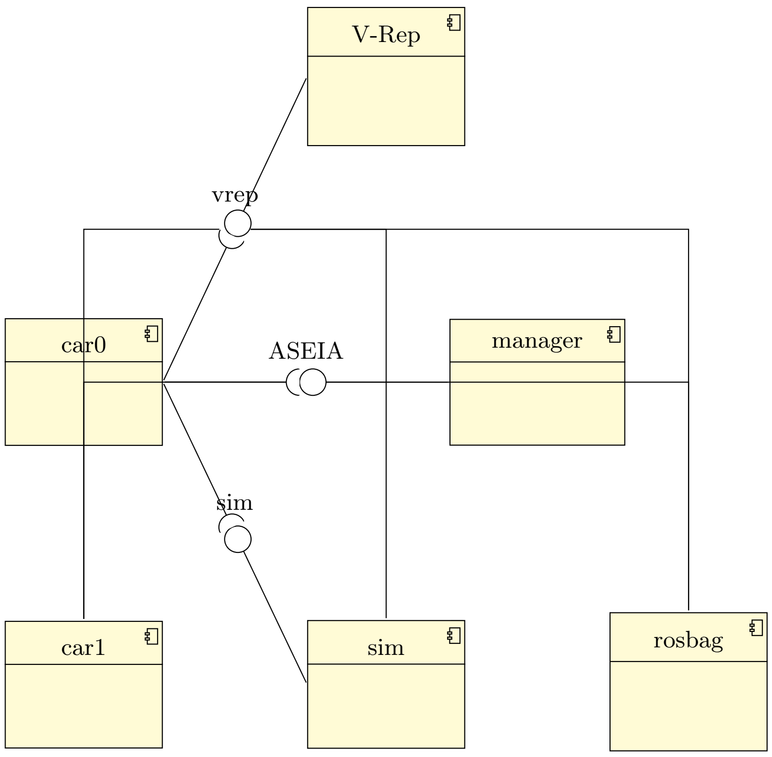


Figure 6.11.: UML Component diagram visualizing the generic architecture of the simulation used for the different evaluations of the car scenario.

a configurable seed to enable reproducibility. The noise is configured using the σ of the Gaussian-distribution and a parameter α by indicating the confidence used to convert the noise to sensor uncertainty. The used virtual distance sensor additionally defines the uncertainty of the output information based on the distance to the front car. In case the distance is below 90% of the maximum distance of the virtual sensor, the uncertainty is statically fixated to 1 m. In case the detected distance is larger than 90%, the uncertainty is set to the maximum distance. This is reasonable as the distance to a potential front car is unknown when the sensor outputs its maximum distance, as the other car may be outside of the detection area of the sensor. Only if a car is detected, the uncertainty can be lessened as the sensor is guaranteed to cover the road in front as far as the detected object.

The available sensors are scarce on purpose to force the system to use *Transformations* to infer the missing information. The ACC behavior is implemented as *PD* controller on the distance information resulting in speed commands. The *Lane Following* behavior is also implemented as *PD* control on the current lane position of the car's resulting steering commands. The control and actuation implementation of the `car` nodes are fully decoupled from ASEIA, as only the virtual sensors publish and subscribe *ASEIA Events*. The scenario uses the following ASEIA attributes:

Position of the car in the simulation scenario as triples of x, y, z coordinates or as road coordinates as r, l, o, with r indicating the road, l indicating the lane offset and o indicating the offset towards the beginning of the road.

Time stamp of the information in simulation time.

PublisherID encoding the source of the information.

Object indicating the car the information is related to.

Orientation of the car as 3 dimensional vector in the same coordinate system as **Position**.

Distance as headway to the next car in front of the current car.

Speed as absolute speed of the car.

The two different coordinate systems of **Position**, **Orientation**, **Speed** and **Distance** enable the computation of the resulting events in two separate coordinate systems specified by the subscriber. This enables the evaluation of ASEIA *Attribute Transformations* in complex scenarios.

The attributes are used to create an *Event Hierarchy* visualized in Figure 6.12 to be used by the `car` nodes and *Transformations.*

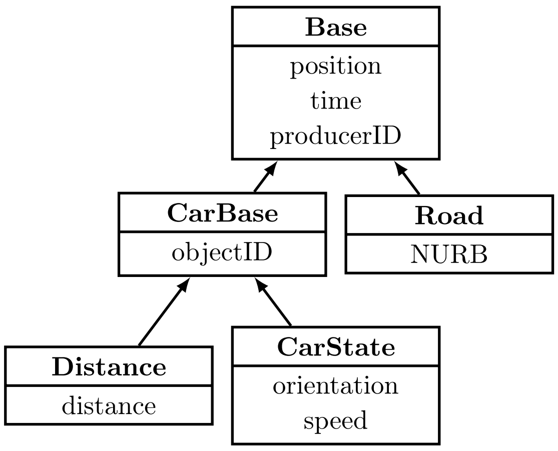


Figure 6.12.: Visualization of the used *Event Hierarchy* of the car evaluation scenarios.

The *ROS* communication used to control and manage the simulation uses *TCPROS* as protocol, whereas, the ASEIA communication uses *UDPROS* as transport layer. This enables a simple separation of data streams between simulation control and ASE delivery. Additionally, UDP communication enables direct emulation of network parameters such as delay and packet loss for evaluation purposes. This network error injection is done using the Linux *Traffic Controller(TC)*[114], which provides a network emulation module *netem* providing support for delaying packets and dropping them with a certain probability. To use this emulation module, the *car* nodes are put into special *Control Groups (cgroups)* (similar to the approach by Handigol et al. [78]) enabling their network output to be modified based

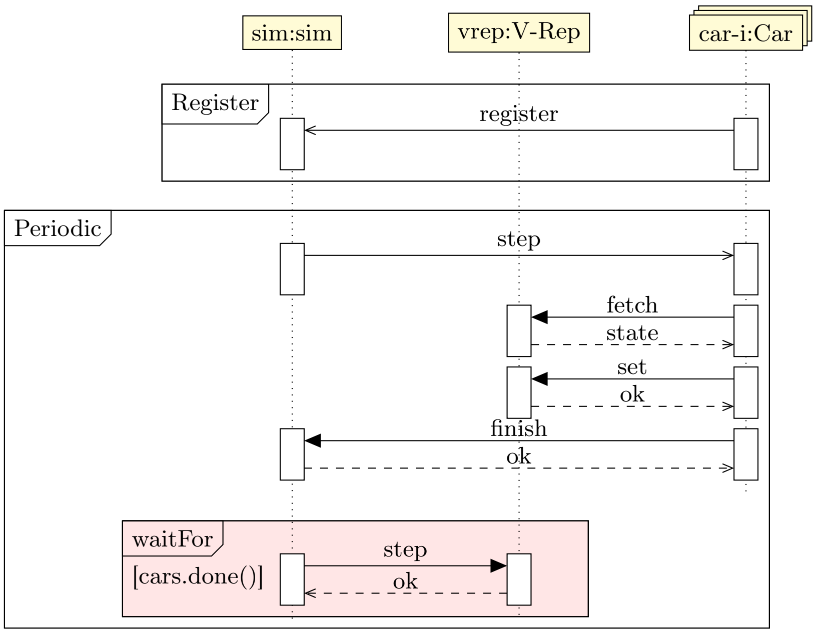


Figure 6.13.: UML Sequence Diagram visualizing the lock step update mechanism to synchronize `car` nodes and *V-Rep*.

on defined *Traffic Control (tc)* network performance emulation rules. The used rules depend on the test case, which is discussed in Section 6.3.3.

The lock-step mechanism is shown in Figure 6.13 as an UML-Sequence diagram. On simulation start, all `car` nodes register themselves with `sim` node. In each simulation cycle, the `sim` uses a *ROS*-Message to indicate the next step to all cars. Each `car` node fetches the current sensor information from the *V-Rep*-Simulation and runs its internal virtual sensor, controller and actuator behavior. Finally, it updates the simulated actuators in the *V-Rep* simulation and indicate the end of their step to the `sim` node. After all cars are done, the `sim` node activates the next step in the *V-Rep* simulation. This mechanism enables the `car` nodes to be updated in sync with *V-Rep*, but in parallel towards each other to mimic a distributed system.

The acquisition of the results of the simulation runs is done by the `record` node, which uses *rosbag* to subscribe to all *ASEIA Events*. These events are fetched using the *TCPROS* transport layer to guarantee the transmission. The CPU utilization of the executed *Transformations* is queried and analyzed using the *perf* [57] framework. The acquisition of performance information is executed system-wide to enable the inference of used CPU resources in comparison with the other components of the system and to also track time spent in *kernel* mode. It also enables a dedicated performance analysis of the different executed *Transformations* within the `manager` node.

6.3.3. Distributed Virtual ACC Sensor

The first scenario aims to use ASEIA to implement an extended ACC behavior in the simulation cars. To this end, three transformations are necessary: *Virtual ACC Sensor Transformation* (`VirtACC`), *ACC Sensor Kalman Fusion* (`VirtACCKalman`) and *UTM to Road Coordinate Transformation* (`UTMToRoad`).

The `VirtACC` *Transformation* uses two **State** events as input to compute the current **Distance** between the objects related to the events. To this end, multiple events need to be stored to enable a matching of the events regarding the **ObjectID** and the **Time** of the events. This *Event Storage* uses an associative map sorting the incoming *Events* according to **ObjectID** and **Time**. In the evaluated test cases the size of this storage was set to 10 *Events* per **ObjectID**. Only if the **ObjectID** is different and the **Time** is equal, a **Distance** computation can be executed. However, these two filters are not enough. Additionally, the **Orientation** of the car needs to be checked against the **Orientation** of the **Distance** to prevent the `VirtACC` *Transformation* to output **Distances** to the back of the car.

The Kalman *Transformation* enables a minimization of uncertainties as it uses knowledge of the system designer and redundancy in the information. According to the specification of the *Kalman Transformation* in Section 4.5.8, it is activated when the subscriber specifies a filter expression containing an upper bound on the uncertainty of the **Distance** attribute (in this case). The used upper bound in the test cases is 0.9 m to limit the output of results to *Events*, which are better then the locally produced *Events*. Depending on the *Scale Reference* specified by the subscriber, an additional *Transformation* is used, which transforms the *UTM* coordinates and orientations acquired from the simulation to *Road* coordinates. This enables a more complex *Transformation Graph*, which might create better results. The resulting *Transformation Graph*s are shown in Figures 6.14 and 6.15.

Table 6.3 shows the configuration of the different experiments. Each test case has been running and recording for 10 minutes. The generic α confidence probability used in all test cases is 0.99.

Name	Ext. ACC Type	Pos. σ	Delay [Min., Max.] in ms	Event Loss in %
Basic	none	0	[0, 0]	0
Ext	UTM	0	[0, 0]	0
Pos1	UTM	1	[0, 0]	0
Pos2	UTM	2	[0, 0]	0
Lossy	UTM	0	[0, 0]	30
Late	UTM	0	[40, 400]	0
Bad	UTM	0	[40, 400]	30
ExtR	Road	0	[0, 0]	0
Pos1R	Road	1	[0, 0]	0
Pos2R	Road	2	[0, 0]	0
LossyR	Road	0	[0, 0]	30
LateR	Road	0	[40, 400]	0
BadR	Road	0	[40, 400]	30

Table 6.3.: Table showing the different configurations of the experiments used to evaluate the car scenario.

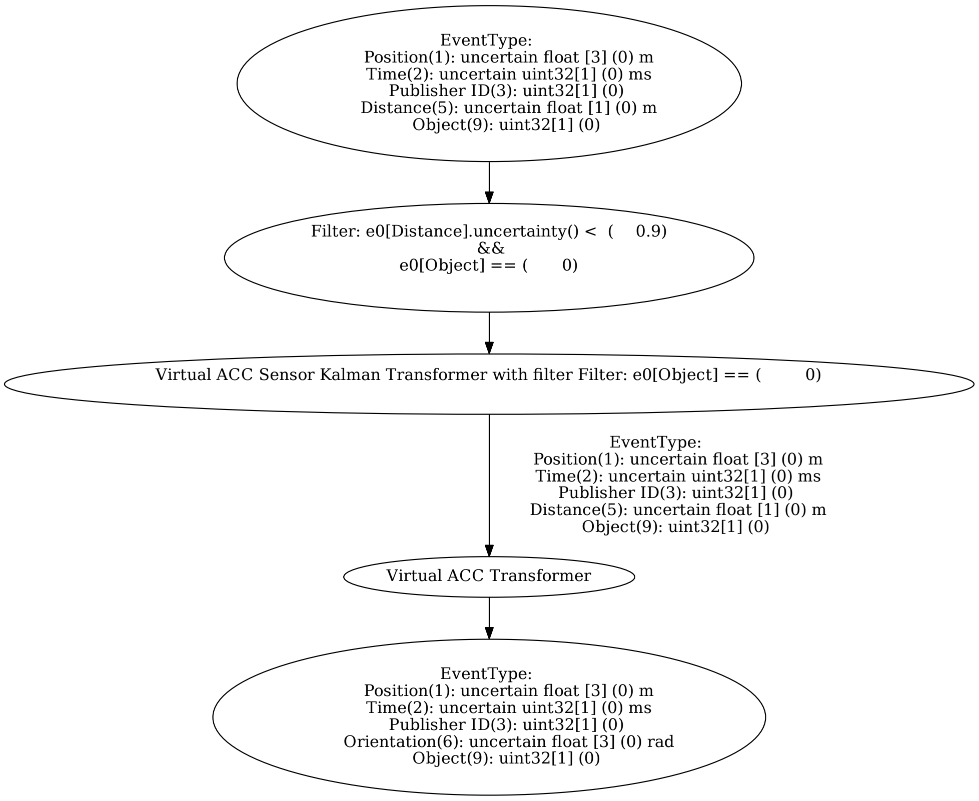

EventType:
Position(1): uncertain float [3] (0) m
Time(2): uncertain uint32[1] (0) ms
Publisher ID(3): uint32[1] (0)
Distance(5): uncertain float [1] (0) m
Object(9): uint32[1] (0)
Filter: e0[Distance].uncertainty() < (0.9)
&&
e0[Object] == (0)
Virtual ACC Sensor Kalman Transformer with filter Filter: e0[Object] == (0)
EventType:
Position(1): uncertain float [3] (0) m
Time(2): uncertain uint32[1] (0) ms
Publisher ID(3): uint32[1] (0)
Distance(5): uncertain float [1] (0) m
Object(9): uint32[1] (0)
Virtual ACC Transformer
EventType:
Position(1): uncertain float [3] (0) m
Time(2): uncertain uint32[1] (0) ms
Publisher ID(3): uint32[1] (0)
Orientation(6): uncertain float [3] (0) rad
Object(9): uint32[1] (0)

Figure 6.14.: Transformation Graph used to provide ASEs for the *Extended ACC* in *UTM* coordinates.

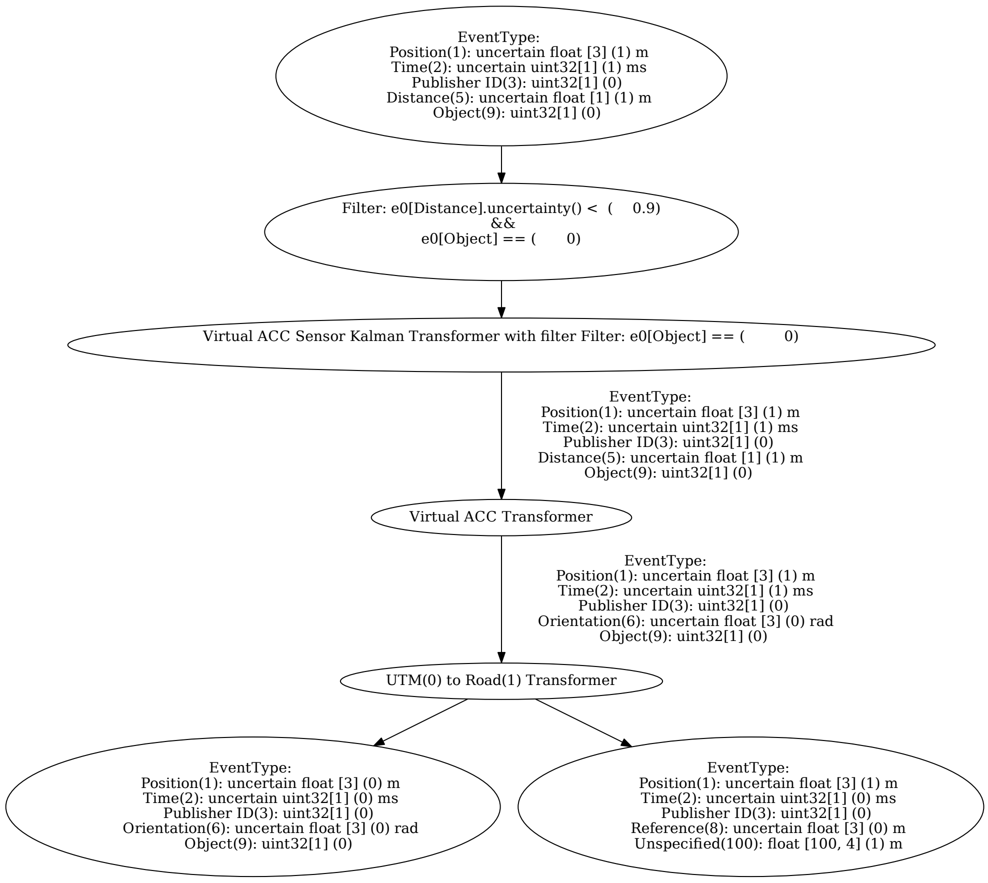

EventType:
Position(1): uncertain float [3] (1) m
Time(2): uncertain uint32[1] (1) ms
Publisher ID(3): uint32[1] (0)
Distance(5): uncertain float [1] (1) m
Object(9): uint32[1] (0)
Filter: e0[Distance].uncertainty() < (0.9)
&&
e0[Object] == (0)
Virtual ACC Sensor Kalman Transformer with filter Filter: e0[Object] == (0)
EventType:
Position(1): uncertain float [3] (1) m
Time(2): uncertain uint32[1] (1) ms
Publisher ID(3): uint32[1] (0)
Distance(5): uncertain float [1] (1) m
Object(9): uint32[1] (0)
Virtual ACC Transformer
EventType:
Position(1): uncertain float [3] (1) m
Time(2): uncertain uint32[1] (1) ms
Publisher ID(3): uint32[1] (0)
Orientation(6): uncertain float [3] (0) rad
Object(9): uint32[1] (0)
UTM(0) to Road(1) Transformer
EventType:
Position(1): uncertain float [3] (0) m
Time(2): uncertain uint32[1] (0) ms
Publisher ID(3): uint32[1] (0)
Orientation(6): uncertain float [3] (0) rad
Object(9): uint32[1] (0)
EventType:
Position(1): uncertain float [3] (1) m
Time(2): uncertain uint32[1] (0) ms
Publisher ID(3): uint32[1] (0)
Reference(8): uncertain float [3] (0) m
Unspecified(100): float [100, 4] (1) m

Figure 6.15.: Transformation Graph used to provide ASEs for the *Extended ACC* in *Road* coordinates.

The *Basic ACC* provides a baseline analysis of the simulated behaviour of the cars. It only uses locally equipped sensors on the car to control the **Speed** regarding the **Distance** to the front. The other experiments use one of the two automatically generated *Transformation DAGs* including computed information from all available cars' sensors. The *UTM* experiments generate the *UTM Transformation Graph*, as shown in Figure 6.14, whereas, the *Road* experiments generate and execute the *Road Transformation Graph*, as shown in Figure 6.15. The difference between the two *Transformation Graphs* is the additional `UTMToRoad` transformation used to transform the original *UTM* coordinates to *Road* coordinates using a special **NURB** event.

Listing 6.3 Implementation of the used *Event Hierachy* for the automtive scenario using the ASE specification DSL

```
1  struct BaseConfig : public BaseEventConfig {
2        using TimeValueType     = Value<uint32_t, 1>;
3        using TimeScale         = Scale<1,1000,0>;
4  };
5  using ObjectID   = Attribute<Object     , Value<uint32_t, 1, 1, false>,
6                               Dimensionless              , Scale<1,1,0>>;
7  using Ori        = Attribute<Orientation, Value<float    , 3, 1, true>,
8                               Radian                     , Scale<1,1,0>>;
9  using Speed      = Attribute<Speed      , Value<float    , 1, 1, true>,
10                              decltype(Meter()/Second()), Scale<1,1,0>>;
11 using Distance   = Attribute<Distance   , Value<float    , 1, 1, true>,
12                              Meter                      , Scale<1,1,0>>;
13 using PoseEvent  = BaseEvent<BaseConfig>::append<ObjectID>::type::
14                                           append<Ori>::type;
15 using StateEvent = PoseEvent::append<Speed>::type;
16 using DistEvent  = PoseEvent::append<Distance>::type;
```

Listing 6.3 shows the implementation of the used *Event Hierarchy* of the automotive scenario. Lines 1–4 show the definition of the **Base** event configuration. In this case the generic configuration is overwritten with the time context being stored as a single unsigned 32-bit integer with uncertainty in ms. The used attributes are defined in Lines 5-12. The ASEs are constructed as a tree of **Pose** and two branches **State** and **Distance**. All events are identified by their attribute combinations (*Event Schemes*). The names are defined for convenience of human developers. The **State** publishers are similar to the **Angle** publisher of the robotic scenario. However, this example shows the integration of the publisher in a class enabling embedding of the P/S extension in existing software designs. Additionally, more data needs to be specified as the **State** sensor data consists of position, orientation and speed. Consequently, all this data needs to be supplied including uncertainty information. The context of the event changed because more uncertainty information is contained.

Listing 6.5 shows the combination of a publisher and a subscriber in the same program. In this specific case the publisher and subscriber share the same ASE definition. Line 3-4 and Line 7-10 define a filter expression used to select beneficial events. In this case the goal event needs to have a distance uncertainty of less than 0.9 m and needs to relate to the current

Listing 6.4 Partial source code of the **State** publisher in the automotive scenario

```
 1 class StateSensor {
 2     SensorEventPublisher<PoseEvent> mPub;
 3     Object mCarBody;
 4   public:
 5     ...
 6     virtual bool update() {
 7       StateEvent mEvent;
 8       const Object::Position pos = mCarBody.position();
 9       const Object::Orientation ori = mCarBody.orientation();
10       const Object::Speed  speed = mCarBody.speed();
11       auto& time  = mEvent[Time()];
12       auto& pos   = mEvent[Position()];
13       auto& ori   = mEvent[Orientation()];
14       auto& speed = mEvent[Speed()];
15       auto& pubID = mEvent[PublisherID()];
16       auto& objID = mEvent[ObjectID()];
17       pubID.value()(0,0) = { mPub.nodeId() };
18       objID.value()(0,0) = { car.index() };
19       time.value()(0,0)  = { getTime(), getDT() };
20       speed.value()(0,0) = { speed, 0 };
21       pos.value()        = {{{pos[0], 0.2}},
22                             {{pos[1], 0.2}},
23                             {{pos[2], 1  }}};
24       ori.value()        = {{{ori.x(), 0}},
25                             {{ori.y(), 0}},
26                             {{ori.z(), 0}}};
27       mPub.publish(mEvent);
28       return true;
29     }
30 };
```

car id. The filters are either executed by the subscriber or by the transformation engine, depending on the generated *Transformation Graph*. The structure of the filter needs to be defined on compile time, whereas constants can be expressed dynamically. The dynamic constants are supplied to the subscriber in ines 18–20. This enables a limited configuration of the filter expression. A structural change of the filter expressions requires the construction of a new subscriber. The received ASEs provide access to their attributes using `operator[]`, which follows the associative map data type of *C++*. This enables a concise and easy to use API to access the contained attributes. Additionally, the attributes provide native linear algebra operations automatically modifying scale and unit. Lines 14–25 enable the computation of the minimum distance and oldest contained time stamp using the attributes of the received ASE. The time value is used by the CPS application to decide on the usability

Listing 6.5 Source code of the **Distance** publisher and subscriber for each car in the automotive scenario.

```
class ACCSensor {
    DistEvent mEvent;
    using ObjectComp = decltype(DistEvent[Object()]);
    using UTMACCUComp = decltype(DistEvent[Distance()].uncertainty());
    VisionDepthSensor mSensor;
    const float mMaxDist = 100;
    UTMACCUComp c = {0.9};
    ObjectComp  o;
    using FilterExpr = decltype(uncertainty(e0[Distance()]) < c &&
                                          e0[Object()] == o     );
    decltype(DistEvent[Time()]) recvTime;
    decltype(DistEvent[Distance()]) recvDist;
    SensorEventPublisher<DistEvent> mPub;
    SensorEventSubscriber<DistEvent, FilterExpr> mSub;
  public:
   ACCSensor(const std::string& path, const Car& car)
    : mSensor(car),
      o(car.index()),
      mSub(&VisionDistanceSensor::handleEvent, this,
           uncertainty(e0[Distance()]) < c && e0[Object()] == o)
    {
        mEvent[PublisherID()].value()(0,0) = { car.index() };
        mEvent[Object()].value()(0,0) = { car.index() };
        ...
    }

    void handleEvent(const DistEvent& e) {
      recvDist = e[Distance()] - e[Distance()].uncertainty();
      recvTime = e[Time()] - e[Time()].uncertainty();
    }

    virtual bool update() {
      auto& time  = mEvent[Time()];
      auto& dist  = mEvent[Distance()];
      time.value()(0,0) = { now, getDT() };
      auto& value = mSensor.distance();
      if(value > mMaxDist*0.9)
        dist.value()(0,0) = { mMaxDist/2, mMaxDist/2};
      else
        dist.value()(0,0) = { value, 1};
      mPub.publish(mEvent);
      ...
    }
};
```

of the received distance data. Depending on the age the application either used the locally generated information or the remote information.

Listing 6.6 Parital Source code of the **Position** to **Distance** *Virtual ACC Transformation.*.

```
class VirtACC : public BufferedTransformer {
...
  virtual Events execute(const Events& e) {
    if(e[0][Time()] == e[1][Time()])
      return {};

    const MetaValue& ori = e[0][Orientation()].value();
    MetaValue diff = (e[1][Position()]-e[0][Position()]).value();
    if((ori*diff).sum() < 5)
      return {};

    MetaEvent eOut(out());
    eOut=e[0];
    eOut[Distance()]=diff.norm();
    return {eOut};
  }
};
```

Transformations use a different API as described in Section 5.2.3. The API is defined to look similar to the CPS API, but omits the static type informatin checks. This enables deployment independent off the *Event Format* of the supplied ASEs. This way, the specified *Transformations* are more generic and adapt on run time to the *Event Formats*. Linear Algebra operations are defined on value-uncertainty complex guaranteeing processing of uncertainty, scale, and unit. This mechanism guarantees a maximum amount of checks being executed during the processing to ensure the quality of the output data. After execution of the *Transformation DAG* a final conversion is done to fit the *Event Format* of the result to the requirements of the subscriber. However, as the data is generated on run time no compiler generated error message are possible. Therefore, if errors occur during execution of the operations, an invalid value is generated. In the default configuration these events are suppressed and filtered at the end of the *Transformation DAG*. However, using a special `debug` policy they can still be delivered for debugging purposes.

6.3.4. Results of the *Basic* Evaluation Test Case

The *Basic* test case is used to establish a baseline for the performance of the ACC behavior of the cars. It will be evaluated regarding homogeneity of **Speed** and **Distance** and **Distance** *Uncertainty*. Figure 6.16 shows the results of the *Basic* test case. As visible, the **Speed** was very inhomogeneous with two peaks at $8\,\mathrm{m\,s^{-1}}$ and $12\,\mathrm{m\,s^{-1}}$ and a very wide distribution of speeds in general. The heat map of *Speeds* show areas of different speeds on the road, which is caused by the change in visibility between the following and front car. The homogeneity of the **Distance** is better. However, the **Distance** *Uncertainty* contains two peaks, which are caused by the two possible detection cases. Whenever the following car can observe the front car, the uncertainty is limited to 1 m. However, if the following car cannot see any front car, the uncertainty jumps to ±50 m, as specified for the virtual **Distance** sensor. Consequently, the **Distance** *Uncertainty* heat map shows areas with very low average uncertainty, in which the following car can observe the front car, and areas, in which the average uncertainty varies because of occlusion of the **Distance** sensor of the following car.

In the following sections, relevant results of the *Transformation Graphs* implementing the *Extended ACC* are presented and discussed. For additional results, refer to the appendix.

6.3.5. Speed Results of the UTM Transformation Graph

Speed Results of the *UTM Transformation Graph*

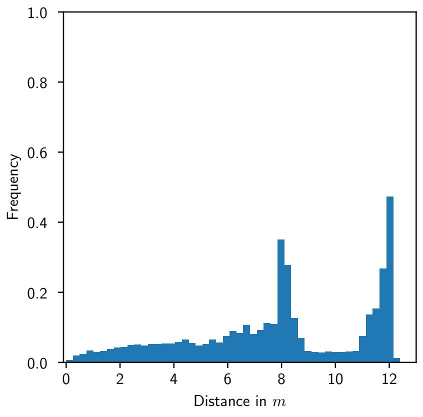
The first result to be analyzed is the distribution of speed over time and position of the following car 0. The goal of the *Extended ACC* is to smooth the driving experience of the passengers. Therefore, the speed should be as homogeneous as possible. Figure 6.17 shows the speed of the following car 0 as histograms and heat maps for the *UTM* scenarios. The histogram enables the evaluation of the homogeneity and the distribution of speed values, whereas, the heat map allows the detection of hot spots on the road, which impact the different ACC cases.

The results shown in Figure 6.17 of the *UTM Transformation Graph* experiments show an increase in homogeneity of speed using the *Extended ACC* in test case *Ext* compared to the local ACC *Basic*. Even if presented with uncertain information or if the delivery is obstructed, the *Transformations* still provide output below the uncertainty threshold of $0.9m$, providing a more homogeneous performance. However, the performance of the *Transformations* is coupled to the quality of the input information and degrades when input quality degrades. Additionally, the homogeneity of speed also degrades. Especially in the *Bad* configuration, inhomogeneous behavior can be observed, as the data was probability delivered too late to be useful to the car and the car often switches to its internal controller often. The heat maps show a decrease in intensity of the hot spots, which indicates a smoother driving. The heat map of *Ext* is nearly hot spot free, which decreases with intensity of injected errors. Even though the *Kalman Transformation* is able to compensate additional uncertainty, the resulting speed is still effected by the injected errors. Even in the worst case of errors of *Pos2* and *Bad*, the heat maps show stronger homogeneity than the *Basic* test case.

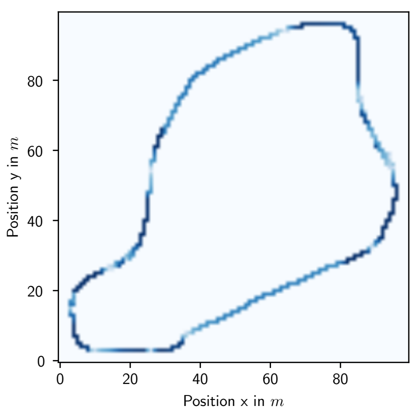
The resulting speed distribution is not fully homogeneous, even in the error-free test case *Ext*. This is caused by offsets of the computed distance at certain spots in the map. This stems from the used Euclidean distance, which under-estimates the real distance of the cars on the track, especially in steep corners. Consequently, the largest inhomogeneities exist between cornering and straight parts of the map.

6.3.6. Speed Results of the *Road Transformation Graph*

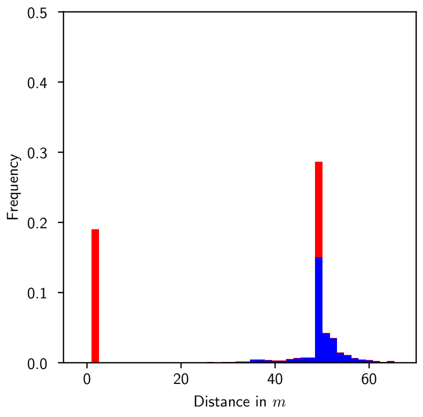
The *Road Transformation Graph* is evaluated similarly to the *UTM Transformation Graph* using histograms and heat maps of **Speed** , see Figure 6.18, visualizing the results of the *Road* test cases.



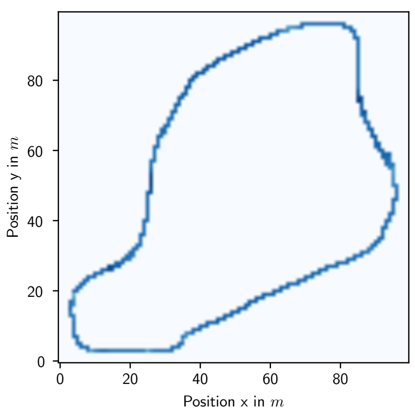
(a) **Speed** Histogram.



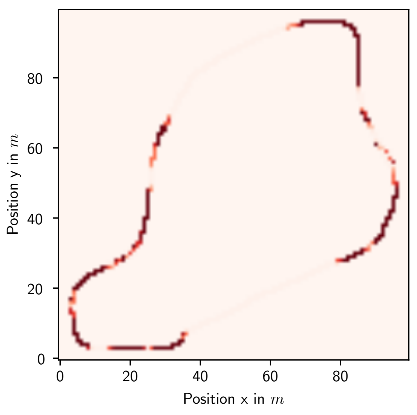
(b) **Speed** Heat Map.



(c) **Distance** Histogram.

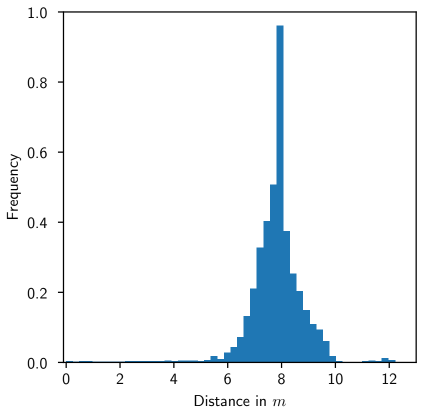


(d) **Distance** Heat Map.

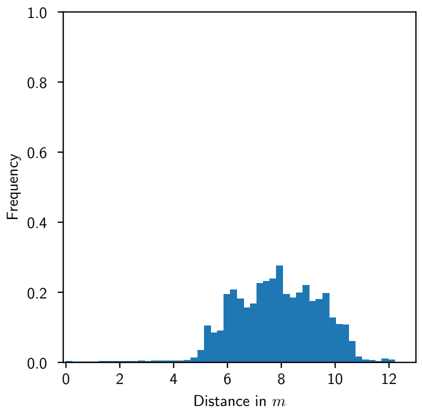


(e) **Distance** Uncertainty Heat Map.

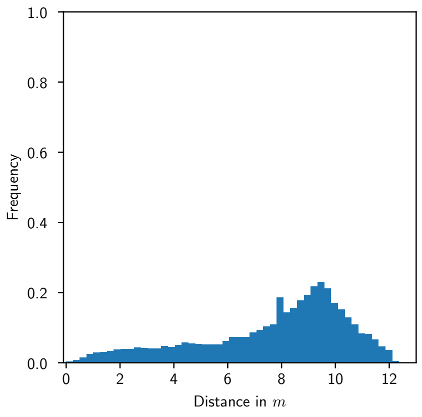
Figure 6.16.: Visualization of result of the *Basic* test case. Heat Maps visualize values of either **Speed**$[0\,\mathrm{m\,s^{-1}}, 12\,\mathrm{m\,s^{-1}}]$ or **Distance**$[0\,\mathrm{m}, 100\,\mathrm{m}]$. Blue colors indicate values, whereas, red colors indicate uncertainty.



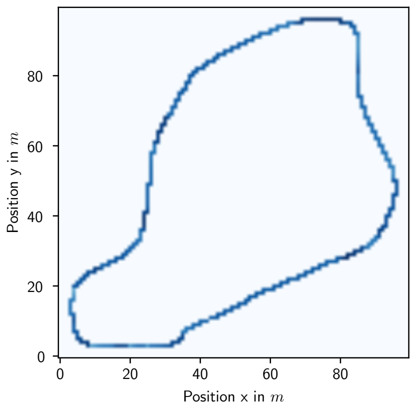
(a) Histogram of Experiment `Ext`.



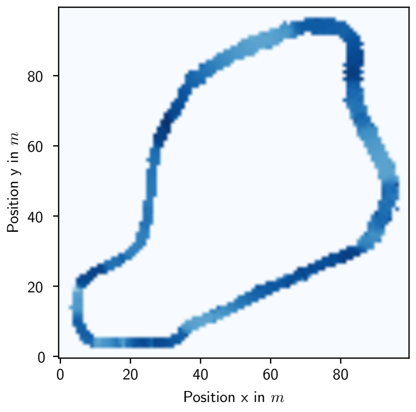
(b) Histogram of Experiment `Pos2`.



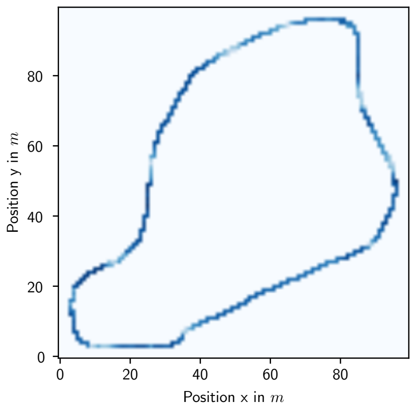
(c) Histogram of Experiment `Bad`.



(d) Heat Map of Experiment `Ext`.

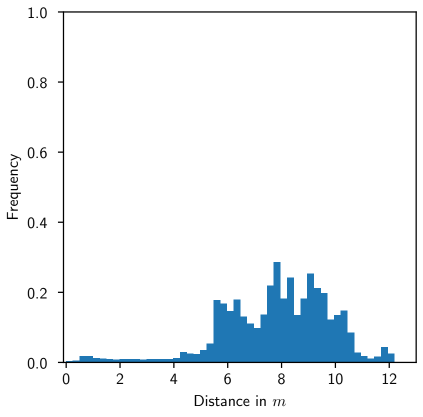


(e) Heat Map of Experiment `Pos2`.

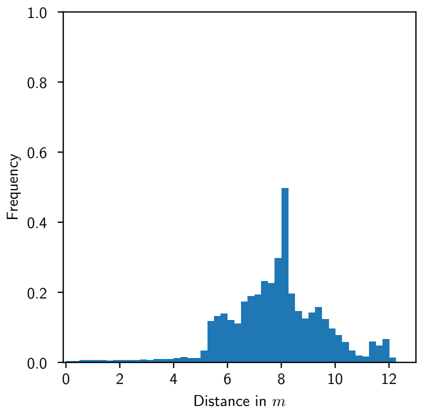


(f) Heat Map of Experiment `Bad`.

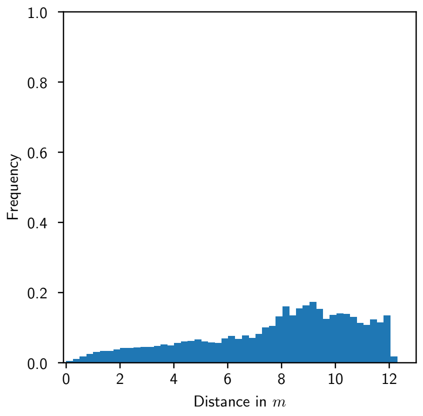
Figure 6.17.: Comparison of the different evaluation scenarios regarding **Speed** of the first car as histograms and heat maps for the *UTM Transformation Graph*.



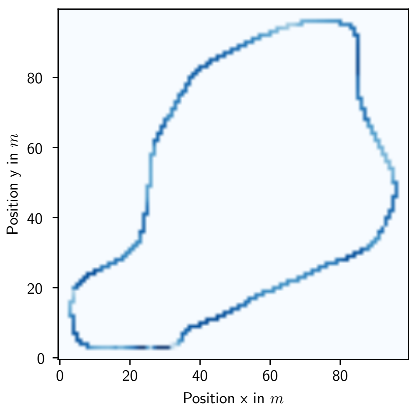
(a) Histogram of Experiment `ExtR`.



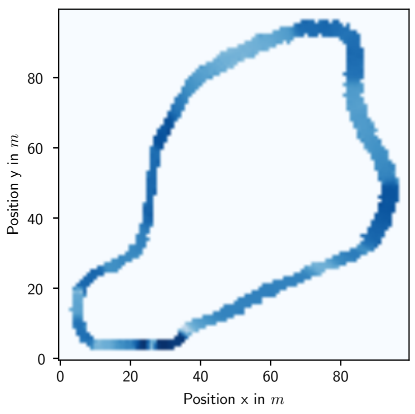
(b) Histogram of Experiment `Pos2R`.



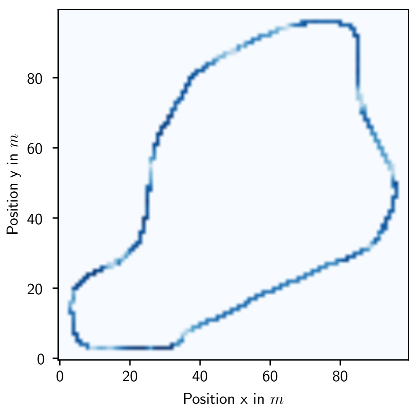
(c) Histogram of Experiment `BadR`.



(d) Heat Map of Experiment `ExtR`.



(e) Heat Map of Experiment `Pos2R`.



(f) Heat Map of Experiment `BadR`.

Figure 6.18.: Comparison of the different evaluation scenarios regarding **Speed** of the first car as histograms and heat maps for the *Road Transformation Graph*.

The *Road Transformation Graph* provides approximately the same homogeneity for the **Speed** as the *UTM* experiments. This presents a contrast to the expectation, which should solve the under-estimation issues at steep corners. However, the additional transformations induce additional uncertainty as more operations are performed on the information. In consequence, the resulting distribution of speed is wider. This is visible in the comparison of `Ext` and `ExtR` histograms. A benefit of the road coordinate system is the used Non-Uniform Rational B-Spline (NURB) description of the road, which adds additional redundancy to the system as it presents additional model information. This mitigates the additional uncertainties in the experiments with bad conditions. Consequently, the maximum performance of the *Road Transformation Graph* is worse, but the average performance is comparable to the *UTM Transformation Graph*.

6.3.7. Distance Results *UTM Transformation Graph*

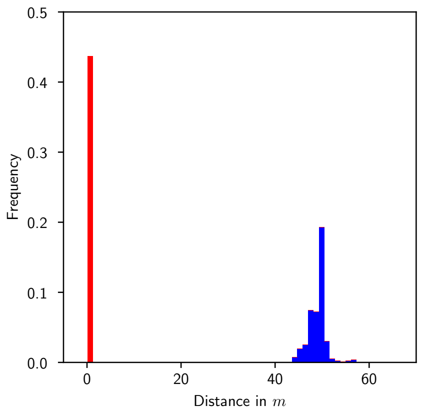
The second parameter to be evaluated is the homogeneity of the virtual ACC **Distance** and the amount of *Uncertainty* assigned to it. The information is visualized as histograms and heat maps, as shown in Figure 6.19.

The distance histogram shows a strong convergence of the observed distances on usage of the *Extended ACC*. Additionally, the used model in the Kalman-Filter heavily decreased the uncertainty. This is caused by the exploited redundancy created through the usage of the local information, which is very accurate, but sometimes unavailable. The remote information, which is generally worse, but is more available. Of course, the quality of the information is decreased, especially in the experiments *Pos2* and *Bad*. In *Pos2* the incoming uncertainty is largely increased, because both cars are injected with additional uncertainty and the used subtraction operation increments the resulting uncertainty by the sum of both uncertainties. The heat map is misleading in this case, as the uncertainty is distributed between multiple positions, because the plot shows the positions as received and used by the cars. There are no hot-spots on the map, because the transformation is not able to grasp the induced offset of its under-estimation of the distance. This is a general limitation of the system, which can only be circumvented using other coordinate systems or other approaches towards the distance estimation. The uncertainty heat maps show a large decrease of uncertainty between *Basic* and all *Extended ACC* text cases. If the input information is worse, the `VirtACCKalman` *Transformation* puts more weight on the local information, which provides precise information in some parts of the road. As the *Kalman* also tracks the distance over time, badly conditioned information from the *VirtACC* may be compensated by the predication step of the Kalman Filter.

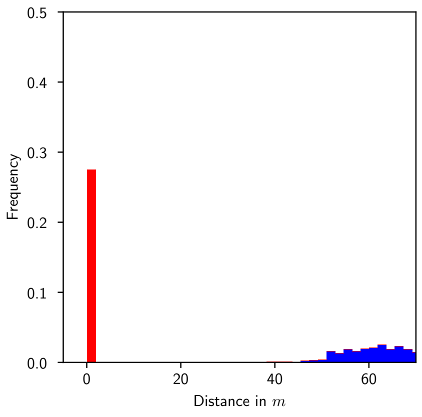
6.3.8. Distance Results *Road Transformation Graph*

The generated distance information of the *Road Transformation Graph* should be very similar to the *UTM* Transformation Graph results as the **Speed** results have already been similar. The evaluation again uses a visualization of the distribution of distance and uncertainty in histogram and heat map (Figure 6.20.

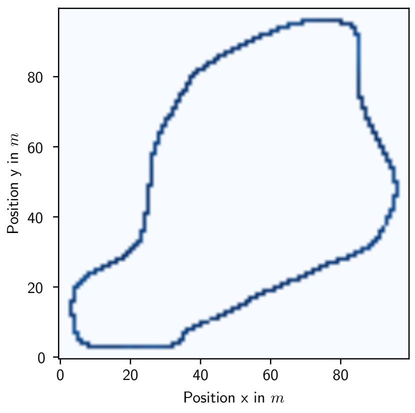
The histogram distribution of the distances is nearly the same compared to the *UTM* experiments. However, the heat maps show two interesting results. The first result is the stronger presence of differences on the heat maps, especially regarding straights and



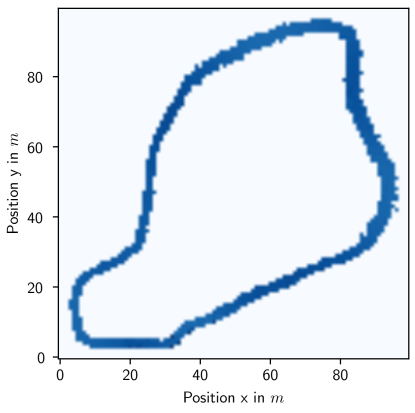
(a) Histogram of Experiment **Ext**.



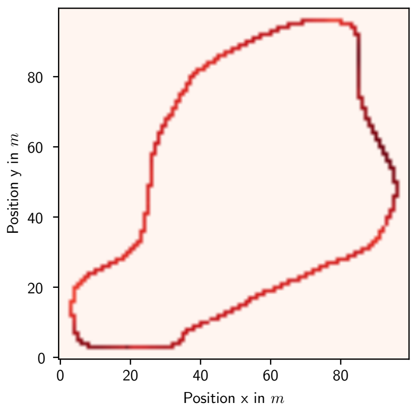
(b) Histogram of Experiment **Bad**.



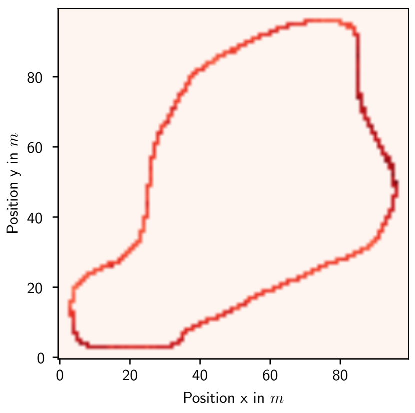
(c) Value heat map Experiment **Ext**.



(d) Value heat map Experiment **Pos2**.

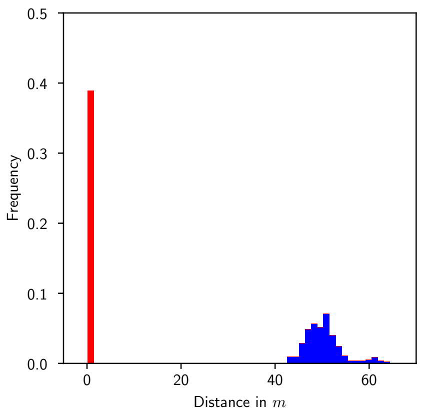


(e) Uncertainty heat map Experiment **Ext**.

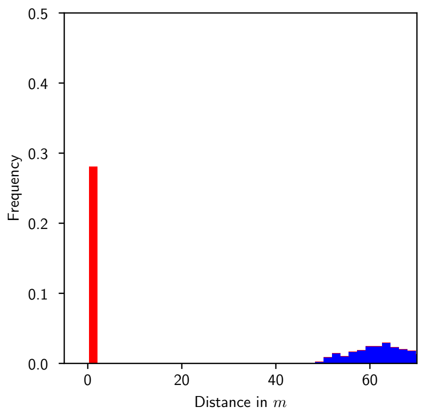


(f) Uncertainty heat map Experiment **Bad**.

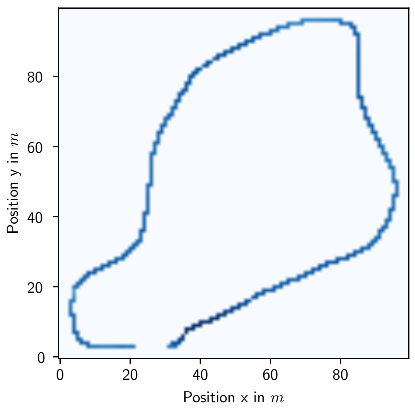
Figure 6.19.: Comparison of the *UTM Transformation Graph* using histograms and heat maps of **Distance** (blue) and **Distance** *Uncertainty* (red).



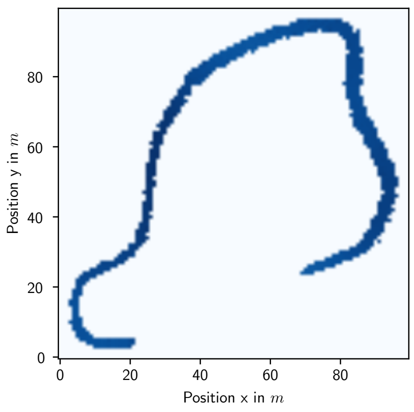
(a) Histogram of Experiment **ExtR**.



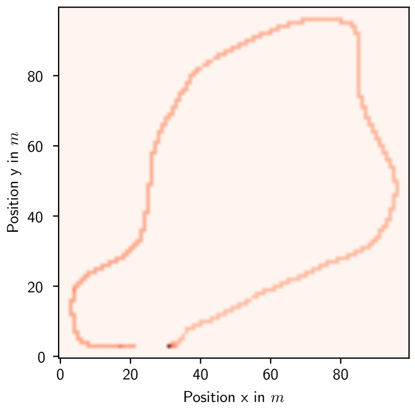
(b) Histogram of Experiment **BadR**.



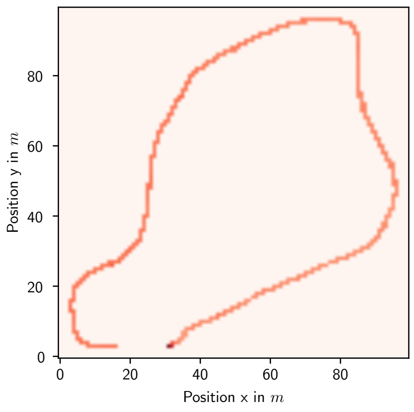
(c) Value heat map Experiment **ExtR**.



(d) Value heat map Experiment **Pos2R**.



(e) Uncertainty heat map Experiment **ExtR**.



(f) Uncertainty heat map Experiment **BadR**.

Figure 6.20.: Comparison of the *Road Transformation Graph* using histograms and heat maps of **Distance** (blue) and **Distance** *Uncertainty* (red).

corners. This stems from the interaction of the remote and the local information. The local information is accurate in whiter areas, whereas, this information is very inaccurate in the darker ares. Consequently, the additional transformation of coordinates create some additional noise, which is stronger in these areas, as the redundant information compensating it is not available. The second observation are the missing spots in the heat maps, which are generated by missing output of the *Road Transformation Graph*. This is caused by a drawback of the road coordinate system, which is the jump occurring at the start/end of the circular road. This jump prevents the `VirtACC` from outputting information with low uncertainty. Consequently, the uncertainty of the events is increased beyond the threshold and the events are discarded. The cars not presented with extended information switch back to their local sensors. The used Kalman-Filter effects the information quality of the road following this section. If the incoming information is worse, the Kalman-Filter takes longer to converge to minimum uncertainty again. This is especially visible in the `Bad` configuration, which contains large gaps in the heat map, where no information was created by the *Transformation*.

6.3.9. Delay Analysis

The third major property of the *Transformation Graphs* is the delay, which is induced by their operations to the delivery of the resulting information. To estimate this, the end of the simulation time stamp interval at arrival is compared to the begin of the time stamp interval contained in the event, which is propagated through the transformations. On each transformation this interval may grow, because newer information is added, but the time stamp of the oldest information defines the lower bound of the interval. The 100 Hz update rate of the simulation is supposed to induce a generic delay of 10ms. Figure 6.21 shows the resulting delays as Box-Whisker diagrams to indicate the mean and distribution of delays for the *UTM Transformation Graph*. As the results were very similar between the *Transformations Graphs* only one is shown. The results of the *Road Transformation Graphs* are available in the appendix.

As visible, the delay of the experiment not inducing network delay is approximately the same. On average it is 10ms with a small deviation as expected by the setup. There are outliers, which can grow up to 100ms. Interestingly, induced noise and induced packet loss rate have similar results on the delay. This may be caused by output events being filtered, because their uncertainty is above the filter threshold for experiments *Pos1* and *Pos2*. This behavior is very similar to any event being lost on route because of the loss rate.

In experiment *Late* and *Bad* the induced delay dominates the packet loss behavior and defines the resulting event delivery delay.

6.3.10. Event Rate Analysis

The rate of publication and the begin of the publication in simulation time is shown in Table 6.4. The entries with **PublisherID** 58 are most interesting, as they indicate the result of *Transformation Graphs*. None of these entries reach 100 Hz. This is caused by the used filter statements of the subscribing cars and the used *Transformations*. For example, a **State** event can only be translated to a **Distance** event when the corresponding **State** event of another car exists. Also visible is the general decrease of event production

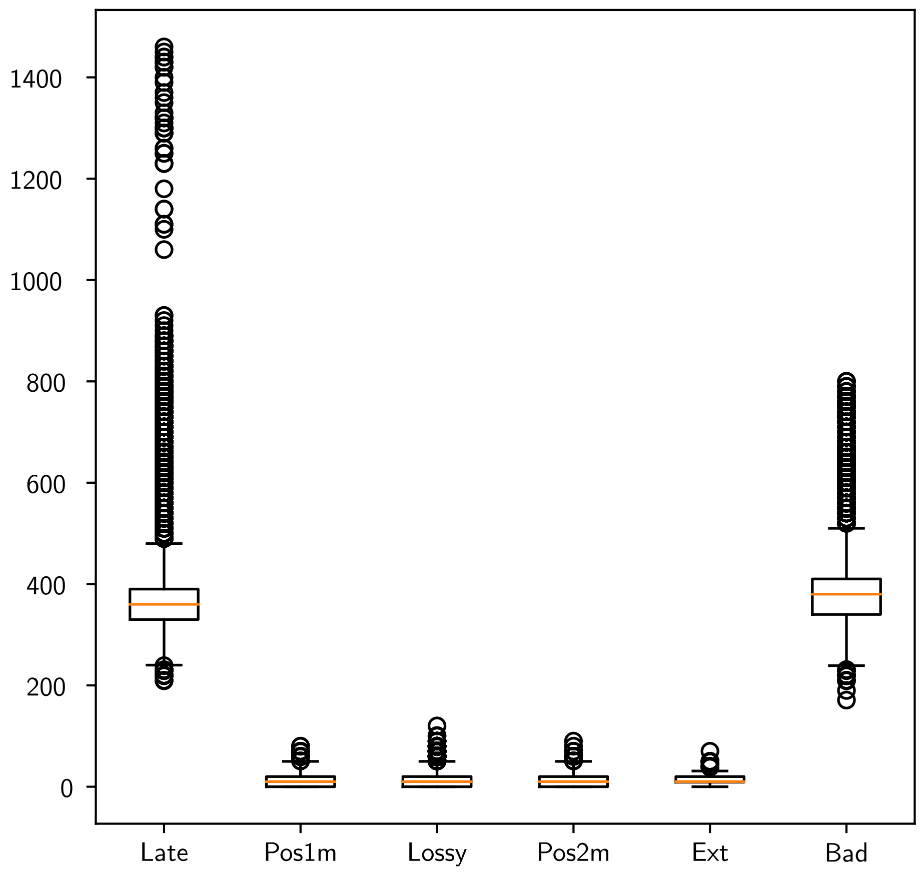


Figure 6.21.: Box-Whisker plot showing the delay induced by the in-network processing for the *UTM* test cases. The delay is measured from atomic event generation until reception of the resulting complex event.

comparing *UTM* and *Road Transformation Graphs*. This behavior is strengthened with bad conditions. However, the quality of the resulting data did not decrease as much, see Section 6.3.8. This stems from the used uncertainty filter, which discards badly conditioned data. Therefore, the system discards information, which the subscriber considered not useful. This increases efficiency with minimal impact on the quality of the delivered data, as only bad data is discarded. This shows a major benefit of the used inherent quality metric and propagation of this metric through the processing. The additionally used road model in the *Road* experiments discards more data when the incoming information is noisy or late, because it provides an additional layer of analysis estimating the uncertainty.

The downside of this behavior is the run time dynamics of the event generation process. In general, it is impossible to know how many events will be generated without knowledge on the amount of publishers and filters used by the subscribers. The information contained in the events might have a large impact on event generation depending on the actually used *Event Filters*.

6.3.11. Performance Analysis

The last attribute to be analyzed is the performance of the event processing in the *Transformation Graphs* within the ASEIA broker. To this end, the CPU utilization is analyzed based on data acquired in the *Ext* and *ExtR* experiments, as they provide the highest event generation rates. Therefore, the resulting data will equally contain all *Transformations*. Table 6.5 shows the CPU utilization of the used *Transformations* within the *Transformation Graph*. The values include the *Event Storage* to enable a correct comparison.

As visible, the `VirtACC` uses approximately 1% of a single CPU on the used Core-i7 5500U for a single car. This is caused by the necessary evaluation of all event combinations of all *ObjectID*s. The `VirtACC` needs two input events, which are stored internally associative according to their **ObjectID**. The *Transformation* needs to check these pairs of events regarding their **Time** and regarding their **Orientation**. This process is time consuming and depends on the buffer size for each **ObjectID**. In this example, the buffer size was set to 10, which mandates the evaluation of 10 event pairs with a rate of 200 Hz. The output of the transformation will always be only a single event representing the minimum distance. The `VirtACCKalman` contributes only slightly to the CPU utilization as it only considers a single event at a time and therefore has an update rate of approxmately 200 Hz (100 Hz induced by the local **Distance** *Events* and 100 Hz induced by the **State** *Events*), which does not depend on the amount of cars in the system. The `UTMToRoad` transformation needs to extract the position of the car along the NURB representation of the road, which poses a minimization problem of a fixed sample size dataset. In the experiments the number of samples for the Road was set to 1000. Depending on the sample size, the performance of the *Transformation* will increase or decrease, but at the cost of additional uncertainty.

Finally, the numbers represent the execution time for the *Transformation Class*, but there are always two instances of the *Transformation Graphs*, because the two cars use different filters containing their respective **ObejctID**. Since the current implementation has no means to share *Sub-Transformation Graphs*, all operations are executed twice. Also, the *ObjectID*-filter cannot be discarded, because the `VirtACCKalman` is state-based and according to the definition of the Kalman-Filter it is a single state tracker.

Experiment	Event	Car	Publisher	Rate in Hz	Startup time in s
Basic	State	0	2	100.00	0.00
Basic	State	1	3	100.00	0.00
Basic	Distance	0	2	100.00	0.00
Basic	Distance	1	3	100.00	0.00
Ext	Distance	0	2	100.00	0.00
Ext	Distance	0	58	94.59	24.69
Ext	Distance	1	3	100.00	0.00
ExtR	Distance	0	2	100.00	0.00
ExtR	Distance	0	58	93.67	0.69
ExtR	Distance	1	3	100.00	0.00
Pos1	Distance	0	2	100.00	0.00
Pos1	Distance	0	58	88.38	27.06
Pos1	Distance	1	3	100.00	0.00
Pos1R	Distance	0	2	100.00	0.00
Pos1R	Distance	0	58	79.82	37.78
Pos1R	Distance	1	3	100.00	0.00
Pos2	Distance	0	2	100.00	0.00
Pos2	Distance	0	58	85.38	23.03
Pos2	Distance	1	3	100.00	0.00
Pos2R	Distance	0	2	100.00	0.00
Pos2R	Distance	0	58	67.23	56.74
Pos2R	Distance	1	3	100.00	0.00
Lossy	Distance	0	2	100.00	0.00
Lossy	Distance	0	58	68.01	19.84
Lossy	Distance	1	3	100.00	0.00
LossyR	Distance	0	2	100.00	0.00
LossyR	Distance	0	58	65.73	25.10
LossyR	Distance	1	3	100.00	0.00
Late	Distance	0	2	100.00	0.00
Late	Distance	0	58	97.97	21.70
Late	Distance	1	3	100.00	0.00
LateR	Distance	0	2	100.00	0.00
LateR	Distance	0	58	91.20	21.10
LateR	Distance	1	3	100.00	0.00
Bad	Distance	0	2	100.00	0.00
Bad	Distance	0	58	68.89	19.89
Bad	Distance	1	3	100.00	0.00
BadR	Distance	0	2	100.00	0.00
BadR	Distance	0	58	65.05	38.58
BadR	Distance	1	3	100.00	0.00

Table 6.4.: Summarizing table of publication rates and startup times of the different *Event Types* produced by the test cases.

Transformation Graph	Transformation	CPU Time in %
UTM	`VirtACC`	0.96
UTM	`VirtACCKalman`	0.12
Road	`VirtACC`	1.02
Road	`VirtACCKalman`	0.12
Road	`UTMToRoad`	0.86

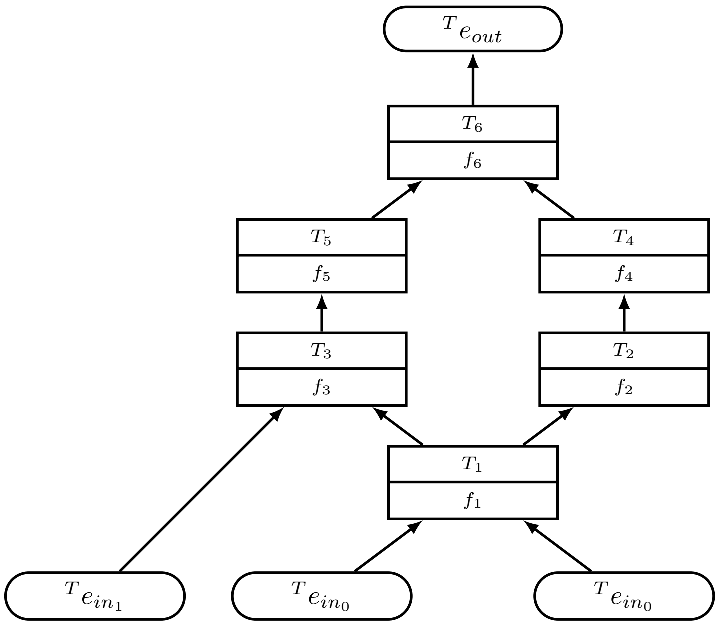
Table 6.5.: Table indicating the CPU utilization of the different *Transformations* within the executed *Transformation Graph* of the *UTM* and *Road Extended ACC without Errors.*

In comparison to the robotic evaluation, as described in Section 6.2, the *transformation Graphs* induce approximately 10 times more CPU utilization, which is caused by the 10 times higher *Event* generation rate of the system. This indicates a strong coupling between *Publication Rate* and CPU utilization of the *Transformations.* The next section analyzes this coupling more thoroughly.

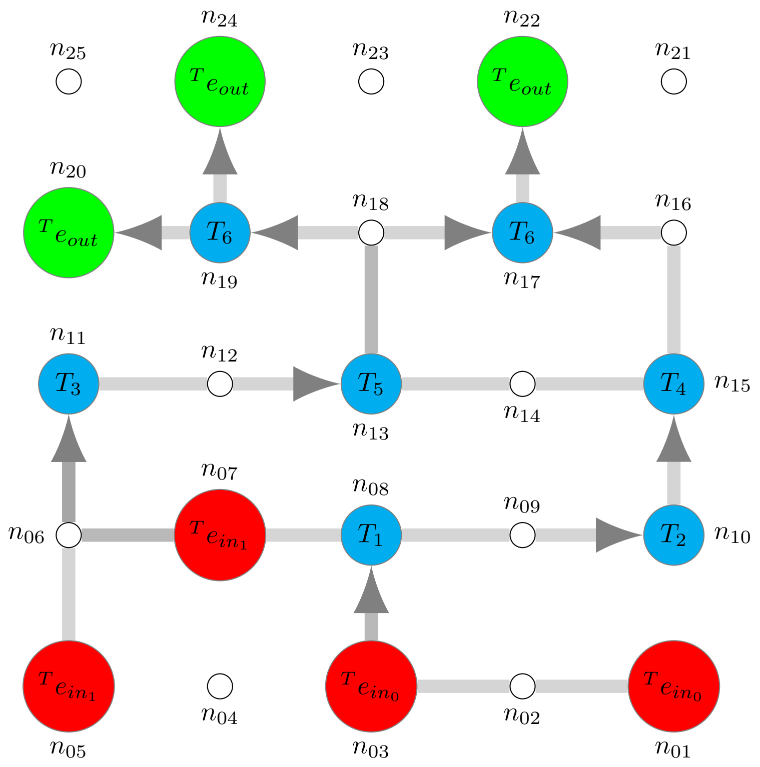
6.4. Scalability Analysis

Scalability is a very important aspect for distributed systems. As ASEIA aims to provide in-network processing in networks of arbitrary size, an analysis of the scalability of generated *Transformation Graphs* to different networks is necessary. As scalability is very difficult to assess in real systems because of the necessary large scale network of nodes and the inherent problems of reproducibility, the evaluation is done analytically. The following section evaluates the *Realization* of *Transformation Graphs* in networks regarding the *Average Normalized Event Transmission Rate* $avg_{T,R}$ for *Transformation Graph* T and *Realization* R. Additionally, a second metric is used: the *Maximum Events to Process* $\#max_{T,R}$. The scalability of the *transformation Graph* under different *Realizations* is analyzed regarding the amount of *ASEIA Brokers* present in the network, as well as the amount of *ASEIA Publishers* and *ASEIA Subscribers.*

Figures 6.22a and 6.22b show an example *Transformation Graph* and its *Realization* in an example network. As visible, the number of producers is higher than the amount of input events of the *Transformation Graph.* Additionally, multiple subscribers exist. Finally, the *Transformations* are distributed into different *Brokers*, which are connected using intermediate nodes and some *Transformations* exist multiple times. Based on the observations of the *Realization*, the *Average Normalized* Event *Transmission Rate* is expressed as a recursive formula according to Equation 6.2, whereas, the *Maximum Events to process* metric is defined in Equation 6.3.



(a) An example *Transformation DAG*.



(b) An example *Realization* of the *Transformation DAG* to an example network.

Figure 6.22.: Visualization of a *Transformation DAG* consisting of multiple *Brokers* executing parts of the graph. The brokers, publishers and subscribers are connected through intermediate nodes only forwarding the *Events*.

$$\#p_j = k_j \sum_{(i,j,h)\in E} \#p_i \tag{6.1}$$

$$avg_{T,R} = \frac{1}{n} \sum_{(i,j,h)\in E} h\#p_i \tag{6.2}$$

$$\#max_{T,R} = \max_{j\in N} \sum_{(i,j,h)\in E} \#p_i \tag{6.3}$$

These equations view the *Realized Transformation Graph* as a graph consisting of n nodes and a set of edges E. The set of nodes N can be subdivided into three disjoint subsets P, S, B containing the publishers, subscribers and brokers respectively. The set of edges is described as a set of three tuples containing the sender, receiver and the distance between them as $E = \{(i,j,h)|i \neq j, i \in P \cup B, j \in S \cup B\}$. To deduce the *Event Transmissions*, the amount of produced *Events* $\#p_i$ in each publisher, broker or subscriber of the network needs to be estimated. Publishers always have an event production of 1: $\forall i \in P, \#p_i = 1$. For brokers this formula 6.1 depends on the amount of incoming *Events* and the *Transformation* specific filter/generation factor k_i. Subscribers use the same formula as brokers, but with their k_i set to 1: $\forall i \in S, k_i = 1$. The *Average Normalized Event Transmission Rate* $avg_{T,R}$ is defined as the sum of all generated *Events* multiplied with the amount of hops they need to travel, divided by the amount of nodes in the graph, see Equation 6.2. The *Maximum Events to Process* $max_{T,R}$ is defined as the maximum amount of incoming *Events* in a single broker, see Equation 6.3.

The results of the two scaling metrics strongly depend on the factor k_i, which is dependent on the actual *Transformation* used. However, a classification of the types of *Transformations* enable the expressions of limits to the factor. Table 6.6 shows this classification regarding the event production. The *Aggregation Unary Filter* and *n-ary Filter Transformation Classes* have implementation specific parameters k, which describe the average event production rate on each encountered input event. For *Aggregation* and *Unary Filter* these k are within the range $[0, 1]$. *n-ary Filters* have a k in the range $[0, B^{n-1}]$, with B being the *Event Storage* size. The value of k depends on the chosen *Policy*, see Section 4.6.5. The *Best* and *Newest Policy* reduces the value compared to the *All Policy*.

The application of the two scalability metrics to the *Realization* of the example *Transformation Graph*, see Figure 6.22, yields the results shown in Equation 6.4 and 6.5 using the assumption of a homogeneous $k_i = k$ for all used *Transformations*.

$$avg = \frac{1}{25}\left(8 + 14k + 14k^2 + 28k^3 + 18k^4\right) \tag{6.4}$$

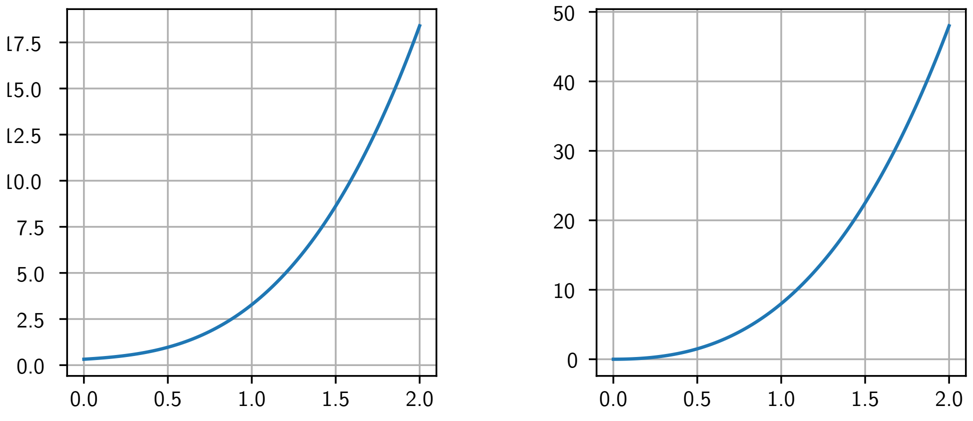
$$max = 2k^2\,(1 + 2k) \tag{6.5}$$

Plotting the exemplary scalability metrics regarding different k in the range $[0, 2]$ yields the Figures 6.23a and 6.23b.

The plots show a strong connection between the parameter k and the amount of *Events* needed to be processed and transmitted in the network. As the generic recursive metric is too complex to be analyzed directly, the following section evaluates the behavior of the *Realization* regarding different generic sub-topologies on different k.

Class	Transformation	#Input Events	#Output Events min	avg	max
Unary	Cast Rescale Interpolation Scale Change Kalman Concatenation Extrapolation	1 2 n	1	1	1
Unary Filter	Constant Selection Unary Process	1	0	k	1
Aggregation	min max avg count median	1	0	k	1
n-ary Filter	Dynamic Selection n-ary Process	n	0	k	B^{n-1}

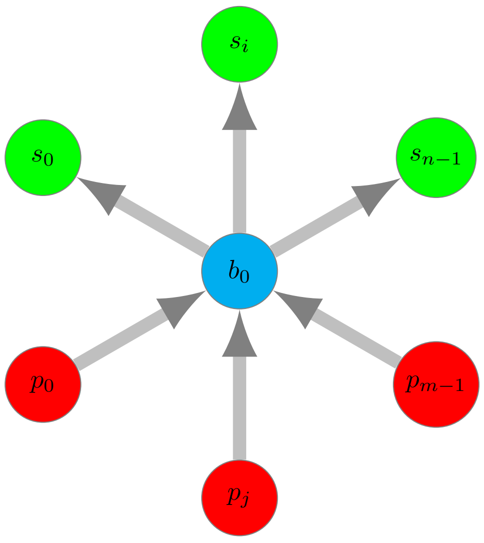
Table 6.6.: Table showing the minimum, maximum and average amount of generated *Events* for each transformed input *Event*.



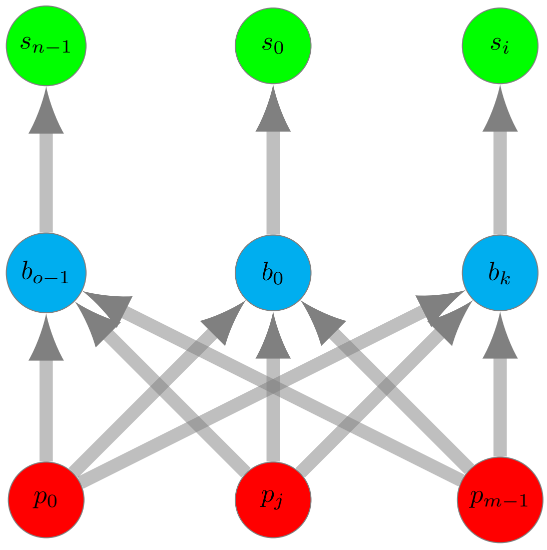
(a) Average Normalized Event Transmission Rate (b) Maximum Events to process

Figure 6.23.: Plots of the scalability metrics regarding the *Realization* of the example *Transformation Graph* dependent on k in the range $[0, 2]$.

To separate the *Transformation DAG* T into single *Transformations* T, the set of publishers P is set to contain all nodes transmitting events to brokers executing T_s, whereas, the set of subscribers S is set to contain all nodes receiving events of T_s. This sub-topology consists of a limited structure which is easier to analyze. There exists two topological extreme cases of the connection between publishers, brokers and subscribers for this sub-topology, which are shown in Figures 6.24a and 6.24b. These differ in the replication of the *Transformation* T_s from the *Transformation Graph* T. In the first case the *Transformation* is executed in a single Broker $B_s = b_0$ located as close as possible to its producers. This maximizes the number of hops to the subscribers, but minimizes the number of hops from the publisher to transmit input events from the publishers. The other topology utilizes multiple Brokers $B = b0, \dots, b_o, o \leqslant |S|$ executing the same *Transformation* closer to the subscribers. This minimizes the hop count for the resulting *Events*, but maximizes the hop count for the input *Events*. Additionally, the amount of *Events* increases as the input *Events* need to be replicated and delivered to all Brokers.



(a) Single Broker Topology to distribute final *Events* to subscribers.



(b) Multi-Broker Topology to distribute final *Events* to subscribers.

Figure 6.24.: Visualization of the two extreme sub-topologies usable to deliver final *Events* produced by a *Transformation Graph* to the subscribers.

The behavior of these topologies regarding the $avg_{T_s,R}$ metric is expressed in the Equations 6.6. The multi-broker topology can be considered as the generic topology. Equation 6.8 describes the behavior of this topology using the parameters H_s^m and H_p^m denoting the average hop count for all used links from any broker to any subscriber and from any publisher to any broker respectively. In the single broker case, the broker number o is 1, which simplifies the formula to Equation 6.8. The maximum multi-broker case is reached when the brokers are embedded in the subscribers, which leads to $H_s^m = 0$. In this case n brokers exist, which leads to Equation 6.9. Assuming an optimal routing algorithm, it can be assumed that $H_P^M \leqslant H_S^m + H_P^m$, $H_S^s \geqslant H_S^m$ and $H_P^s \leqslant H_P^m$. For arbitrary networks H_S^m and H_P^m can be considered as functions of the amount of brokers $|B|$ used.

$$avg_{T_s,R} = \frac{1}{|N|}\left(km \sum_{(i,j,h)\in E, j\in S} h + \sum_{(i,j,h)\in E, i\in P} h\right) \tag{6.6}$$

$$avg_{T_s,R_m} = \frac{|P_s|}{|N|}\left(k|S_s|H_S^m + |B_s|H_P^m\right) \tag{6.7}$$

$$avg_{T_s,R_s} = \frac{|P_s|}{|N|}\left(k|S_s|H_S^s + H_P^s\right) \tag{6.8}$$

$$avg_{T_s,R_M} = \frac{|P_s||S_s|H_P^M}{|N|} \tag{6.9}$$

$$\tag{6.10}$$

6.4.1. Sub-Topology Analysis

In the following, some example topologies are analyzed to extract an approximation of $avgT_s, R$.

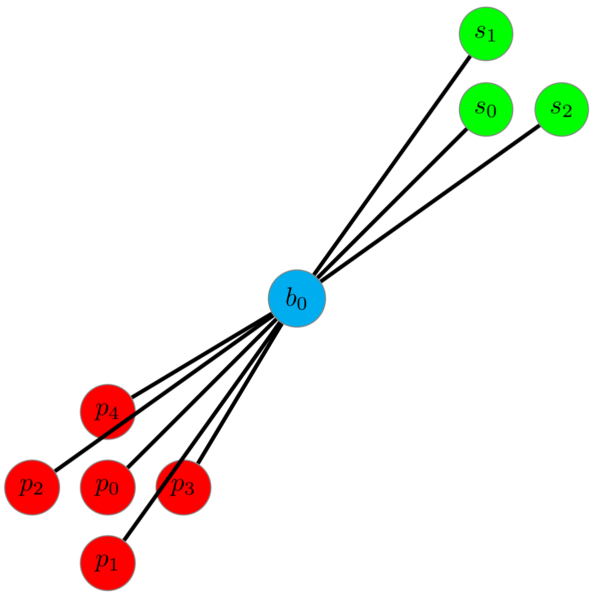


Figure 6.25.: Figure showing an example linear topology of a single cluster of subscribers and publishers and a single broker connecting them.

For a linear topology of publishers and subscribers, as visualized in Figure 6.25, the sum of average hops between publishers and brokers and subscribers and brokers is constant: $H_S(|B|) + H_P|B| = C \approx const$. Consequently, $H_S(|B|)$ can be expressed as $C - H_P|B|$. Inserting the result in Equation 6.7 yields $avg_{T_s,R} \sim k|S|\left(C - H_S(|B|)\right) + |B|H_P(|B|) = H_P(|B|)\left(|B| - k|S|\right) + k|S|C$. The term $k|S|C$ is constant as it is not dependent on $|B|$, which yields $|B| - k|S|$ as the defining term. It follows that minimizing $|B|$ minimizes that term. In consequence, single brokers are best suited for linear topologies. The remaining question is the distance of the broker to the subscribers and the publishers. After $|B| = 1$, the term $H_P(|B|)\,(|B| - k|S|)$ simplifies to $H_P(|B|)\,(1 - k|S|)$. Consequently, the broker should be deployed closely to the publishers if $k|S| > 1$ and closely to the subscribers if $k|S| < 1$. In the case $k|S| == 1$, the placement is not inducing any change.

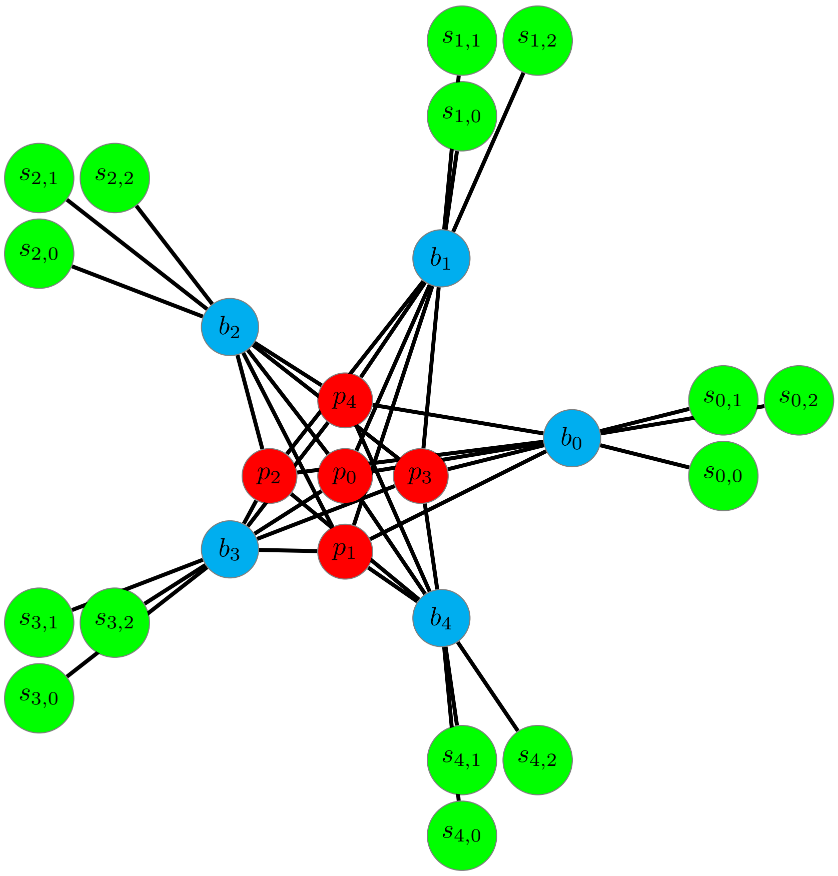


Figure 6.26.: Figure visualizing an example star topology with publishers in the center and single brokers for each arm.

A star topology with publishers in the center, as visualized in Figure 6.26, can be considered to consist of separate independent linear topologies. Therefore, the same rules apply as for a single linear topology. The factor $k|S|$ defines how close the brokers should be to the publishers. A large $k|S|$ moves the broker away from the center towards the subscribers. This behavior is independent for each arm. If multiple arms have small $k|S|$ values, these brokers may be merged in the center to a single centralized broker.

A star topology with subscribers in the center, as shown in Figure 6.27, shows completely different behavior. As all published events need to be delivered to all the brokers, the placement of the brokers has little influence on the $H_P(|B|)$, but a centralized broker minimizes $H_S(|B|)$. In consequence, the term $k|S|H_S(|B|) + |B|H_P(|B|)$ is simplified to $k|S|H_S + |B|H_P$, which is minimized by setting $|B|$ to 1. Therefore, the best configuration in this scenario is a single broker in the center of the star.

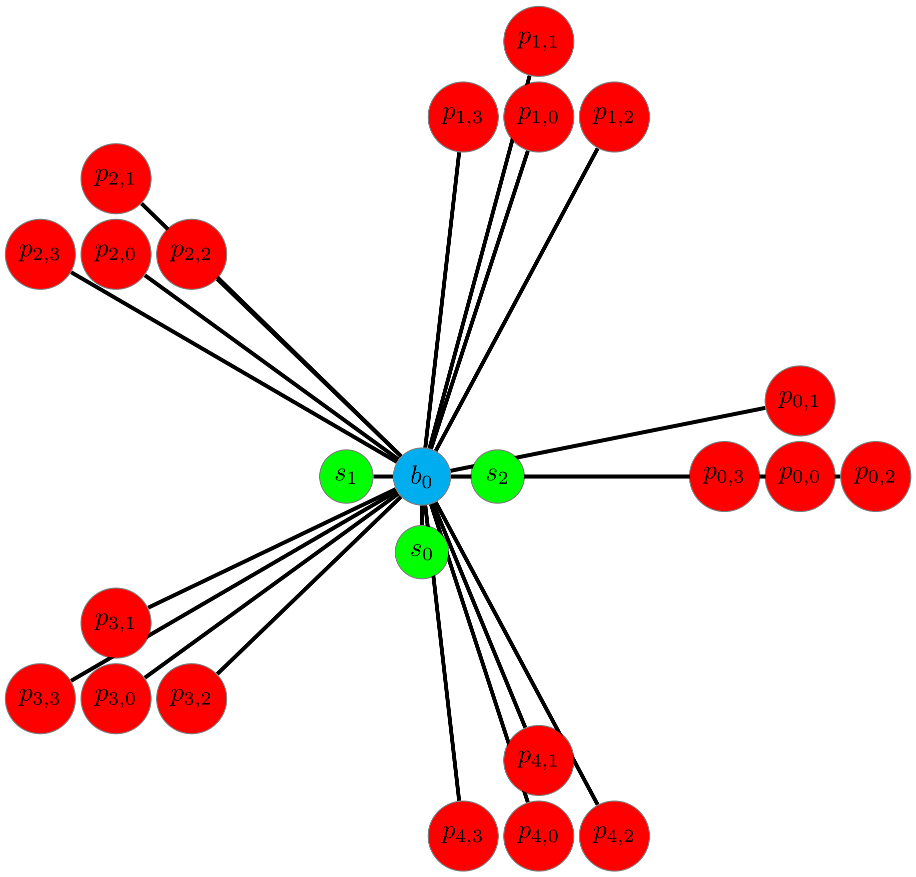


Figure 6.27.: Figure visualizing an example star topology with subscribers in the center and a single broker in the center.

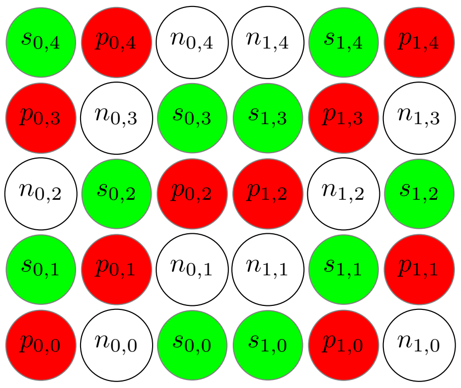

$s_{0,4}$ $p_{0,4}$ $n_{0,4}$ $n_{1,4}$ $s_{1,4}$ $p_{1,4}$
$p_{0,3}$ $n_{0,3}$ $s_{0,3}$ $s_{1,3}$ $p_{1,3}$ $n_{1,3}$
$n_{0,2}$ $s_{0,2}$ $p_{0,2}$ $p_{1,2}$ $n_{1,2}$ $s_{1,2}$
$s_{0,1}$ $p_{0,1}$ $n_{0,1}$ $n_{1,1}$ $s_{1,1}$ $p_{1,1}$
$p_{0,0}$ $n_{0,0}$ $s_{0,0}$ $s_{1,0}$ $p_{1,0}$ $n_{1,0}$

Figure 6.28.: Figure showing an example uniform grid topology without brokers.

The last topology considered is the uniformly distributed two dimensional grid consisting of equally spaced publishers and subscribers. For a square grid of size x, this topology has a small changing $H_P(|B|)$ of approximately $H_P(|B|) = [1/3x, 2/3x]$. On the other hand,

$H_S(|B|)$ depends strongly on $|B|$ as $H_S(|B|) = \frac{1}{3}x\frac{1}{\sqrt{(|B|)}}$ if the brokers are uniformly distributed. The defining term becomes $avg = \frac{k|S|}{\sqrt{(|B|)}} + 2|B|$. The minimum of this formula is $B_{min} = \frac{k|S|}{4}^{\frac{3}{2}}$. Figure 6.29 shows a plot of the number of brokers $|B|$ based on k values and subscriber numbers $|S|$. As visible, even for large k, the optimal number of brokers is only half of the number of subscribers. The *Best* and *Newest Policies* limit k to $k < 1$. Consequently, the optimal broker number for this topology is limited to $|B| < \frac{\sqrt{|S|}^3}{8}$.

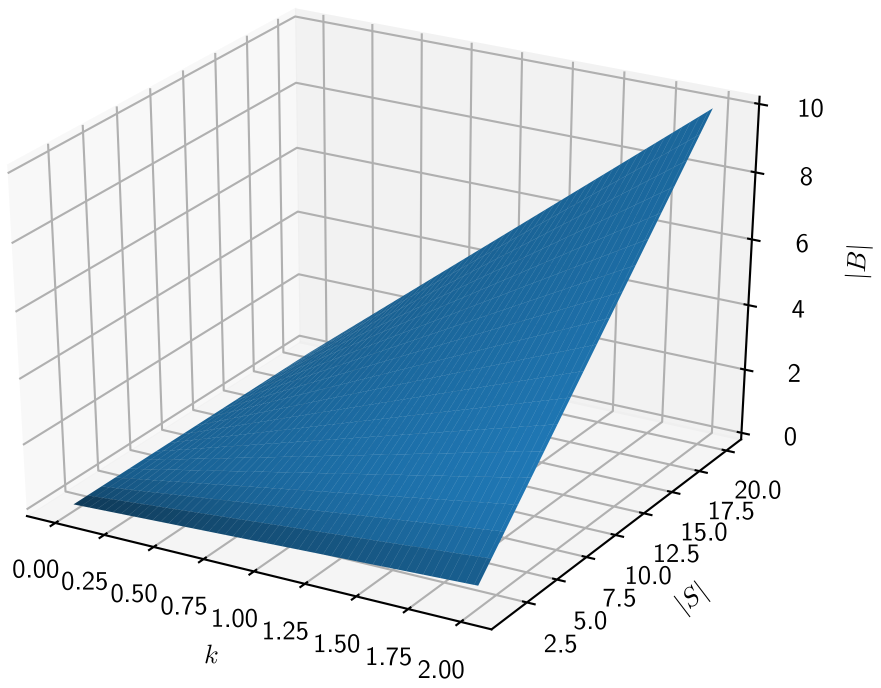


Figure 6.29.: Surface plot of the optimal number of brokers $|B|$ based on the *Transformation*'s filter/generation factor k and the subscriber count $|S|$.

6.4.2. Subscriber and Publisher Scalability

Subscriber scalability depends on the factor $k|S|H_S(|B|)$, because the relevant term of the $avg_{T_s,R}$ equation is $k|S|H_S(|B|) + |B|H_P((|B|)$ and $|B|H_P((|B|)$ can be considered to be constant in case no change in the number of brokers occurs. Therefore, the subscriber scalability is good when the *Transformation* filter/generation factor is small or the average distance to the broker is small. Consequently, *Transformation Graphs* with strong filters discarding many incoming events are positive. Also, topologies with small distances between brokers and subscribers are beneficial in this case.

Publisher scalability only depends on the average publisher to broker distance $H_P(|B|)$, because the factor $|P|$ is a general factor independent of the number and placement of brokers. When a publisher is added, which increases $H_P(|B|)$, additional communication is necessary, but the general resource consumption of the dissemination of the *Events* of this publisher cannot be mitigated by the placement or number of brokers. However, the structure of the *Transformation Graph* has an impact on the performance. When *Transformations* with small k values are generating the input events to the *Transformation* in question, fewer events need to be considered and the effect of a high k value is mitigated. Unfortunately, the movement of *Transformations* within an *Transformation Graph* is an optimization, which is not generally valid and requires further research to establish optimization rules similar to the ones of the relational algebra of databases, see [48] and [47].

6.5. Summary

This evaluation chapter provides four different evaluations. The first evaluation provides insights on the expected quality and scalability of the *Rescale Transformation* of **Time** deployed to WSN. This is an important aspect for the usage of ASEIA in WSN scenarios to compensate for the lack of a global time base and still enable filter expressions and therefore, *Event* inference based on **Time**. The second evaluation considers the usage of ASEIA for robotic navigation using *Transformations* on **OccupancyGrids**, which are directly fed to external navigation software. The results show the flexibility of the system as well as the easy integration of ASEIA with other software, which is important to achieve the goal of dynamic composition of CPS. The third experiment evaluates ASEIA's capabilities to describe and process physical phenomena observed by sensors in mobile entities and automatically process the information. The results show interesting results regarding the propagation of uncertainty, but also indicate scalability problems in certain types of *Transformations*. The last evaluation focuses on theoretical scalability based on regular topologies to evaluate the behavior of the system for larger networks. The results showed the importance of the event generation/filter factor k for the performance and scalability of the system. Additionally, it showed some topologies to provide better performance when brokers are deployed based on the actual k value. A remaining question is the acquisition and dissemination of the k factor on run time to optimize broker placement and usage.

7. Conclusion

This thesis approaches the topic of dynamic composition of CPS, which is a key feature of future applications in the area of Industry 4.0, autonomous cars and autonomous robots. The thesis focuses on the description and dissemination of information flowing between components in CPS to enable an automatic construction and reconfiguration on run time. To this end, a structured description of the information in form of ASE has been developed, which combines the description of data, semantics, uncertainties and contextual attributes. The semantics are described using a vocabulary of elementary attributes specific to a scenario. These attributes may relate to physical phenomena, but do not necessarily need to. The set of attributes known to the system is not run time static, but may be extended by the components of the system as needed. ASE allow individual development and better reuse of existing CPS components because the coupling between them is reduced. The individual components may use ASEs as a specification mechanism of their requirements towards the necessary information. ASE provide means to specify adaptive interfaces between the components. The challenge of these interfaces is the possible lack of appropriate information, which requires the execution of information processing steps to transform existing information to fullfil the requirements of the specifying component. To this end, an in-network information processing system was developed, which is called ASEIA.

CPS using ASEIA contain loosely coupled components, because ASEIA decouples them regarding space, flow, data type and semantics. This eases the development of the component and increases its flexibility. The components may only state their requirements regarding the incoming information, but without knowing the other components of the system providing information. The necessary requirements are specified as *Event Types* and *Filter Expressions* on the events described by the *Event Types*. The thesis provides a filter language to allow the description and dissemination of these requirements through the network.

The processing of information is done within the network disseminating the information through specialized nodes called ASEIA brokers. These execute *Transformations*, which are automatically deployed if they are needed to fullfil a component requirement. The *ASEIA Brokers* act individually and provide communication channels between components in a fully distributed system. Possible *Transformations* are classified and structured by the thesis to enable this automatic deployment and to combine them if necessary. The described *Transformations* provide capabilities to transform the data, types and context of the information. Additionally, sensor fusion operations are supported, which enable a dynamic trade-off between e.g. update rate, observation range and uncertainty. To ease the development of new *Transformation* ASEIA provides facilities to compute the sensor data and the attached context as a single entity. These processing operations are context aware and automatically adapt scales, units and uncertainty of the data. This allows the components to specify requirements on quality attributes like QoC and QoS. The resulting parameters of ASEIA regarding the comparison criteria used in the state of the art analysis of Chapter 3 are shown in Table 7.1. As visible, ASEIA fulfills most of the described requirements. A time coupling may only be observed whenever no buffering is used in the *Transformations*. Additionally, QoS provision depends on the used P/S system. Therefore, it cannot be guaranteed generally.

Criteria		ASEIA
Coupling	Time	(x)
	Space	
	Flow	
	Data Type	
	Semantic	
Context	Time	x
	Space	x
	Uncertainty	x
	Semantics	x
Quality	Service	(x)
	Context	x
Distribution	Type	distributed
Resources	CPU	++
	Memory	+
	Network	++
Filter	Model	Subscription-based
	Elementary	++
	Composite	++
	Sequence	–
Processing	Model	Rule
	Transformation	++
	Aggregation	++
	Fusion	++

Table 7.1.: Table indicating the results of ASEIA regarding the comparison criteria of the state of the art (Chapter 3).

The concepts of ASEIA are implemented in a prototype, which has been evaluated regarding the uncertainty propagation as well as the run time behaviour for real robotic and simulated car scenarios. The concepts were validated and proved to increase flexibility, while still providing good performance. The uncertainty propagation within the *Transformation* enabled the simulated car CPS applications to assess the current state of the incoming information. Additionally, the filter expression system of ASEIA successfully embedded a *Kalman Filter Transformation* to decrease uncertainty based on applications requirements. In the case of unsuitable information because of high uncertainty in the data, the ASE are automatically filtered, which decreases the CPU and network load of the applications, brokers and the communication backbone. The special issue of time attribute synchronization has been tackled by a special time scale transformation, which has been evaluated in simulation and on real WSN hardware. An evaluation of the scalability of the system in different regular network topology types has been performed, which showed a very strong dependence of the system on the placement and number of ASEIA brokers and the event generation/filter factor k, which indicates how many events are generated on average for each incoming event.

The optimization of the broker number and placement within real networks is subject to future work, as multiple parameters such as the average hop count from publisher to broker and the average hop count between broker and subscribers exist. Additionally, the factor k is a pure run time variable, which is very difficult to estimate on design time. Therefore, mechanisms need to be established to estimate this factor on run time. Dissemination of the results and the uncertainty of the estimation through the network is already implemented by ASEIA. The topology analysis already showed efficient broker placement strategies regarding the minimization of ASE count and therefore, network load. However, other metrics such as latency and load balance exists, which may be relevant to the real system. These three metrics are typically in conflict and require a multi-objective optimization on run time as some or all of them depend on k. The description of necessary optimization goals of the components as well as the distributed optimization present a major research challenge. Preliminary work was executed by Zug et al. [171] to optimize the delivery delay of sensor information in a distributed network. Another approach was developed by Pietzuch [122], which aims to minimize delay of event delivery by optimization of operator placement in the network. Future work should aim to combine the individual approaches to provide optimal broker placement based on the systems needs.

Another approach to prevent overload of the network by high event generation counts is early filtering of incoming events. This also decreases the number of events processed in the brokers and prevents processing overload. To this end, the filter expressions stated by the subscribers and *Transformations* may be used. However, the automatic *Transformation DAG* generation disables a prediction of the order of *Transformations* in the graph. To minimize the k factor for the whole graph *Static Selection* statements should be propagated downwards through the *Transformation Graph*, while they adapt themselves to the executed *Transformations*. This requires a filter expression algebra enabling operations on filter expressions, which are linked to operations of the *Transformation Algebra*. The result is the execution of the *Static Selection* filters as early as possible in the *Transformation Graph*, which reduces the k factor, because individual ASE are filtered out early without the need to store and combine them with other ASE.

An extension of this approach would be the integration of automatically generated filter expressions in the *Transformation Graph* when the network or the broker is overloaded. This enables a behaviour similar to *Aurora*, see Section 3.5, and *Solar*, see Section3.6, to trade information and context quality against event loss in a predictable way. The challenge in this extension lies in the acquisition of the overload indication parameters in the brokers distributed in the network. ASEIA *Transformation* mechanisms can be used to fuse together the results of the individual brokers to detect the overload. Afterwards, automatic generation of additional *Static Selection Filter Expressions* may be used to mitigate this overload. Another mitigation strategy would be the change of *Realization* of the *Transformation DAG* to different ASEIA brokers.

ASEIA currently provides a high-level API to state physical processes and mathematical computations on value-uncertainty complexes. This already enables an efficient statement of *Heterogeneous Transformations*. A formal description of the linked algebra between filters and *Transformations* enables the definition of a DSL to specify *Transformations* even more easily. The DSL may provide additional design-time error reporting, which is not possible with the current API-based description of *Transformations*. It also enables an interactive execution of *Transformations* on ASEIA brokers providing human users with an interface to test and tune parameters on run time. Currently, the used LLVM-based interpretation of *Transformation* through intermediate code requires a compile and deploy step, which is cumbersome in development situations, where multiple changes are done to the *Transformation*.

The last possible future optimization of the system is the integration and evaluation of different uncertainty models, which may provide a better estimation of the uncertainty present in the information or the context. Other presentations such as Gaussian-mixture models provide means to estimate the uncertainty with a higher accuracy, but may require much more information on the system in which they are used. This trade-off between uncertainty estimation accuracy and system specification effort is worth future investigation. The current setup of ASEIA enables an easy integration of different uncertainty models, which eases the comparison, but it also allows the combination of different uncertainty models in the same system. This presents further research challenges as the integration and interaction of different uncertainty models is not well researched yet.

Bibliography

[1] ISO 11898-1:2015(E). Road vehicles – controller area network (CAN) – Part 1: Data link layer and physical signalling, December 2015.

[2] IEEE Std 1451.5-2007. IEEE Standard for a Smart Transducer Interface for Sensors and Actuators Wireless Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats, October 2007.

[3] ISO/IEC 14977:1996(E). ISO/IEC Information technology – Syntactic metalanguage – Extended BNF, December 1996.

[4] IEEE Std 1609.3-2016. IEEE Standard for Wireless Access in Vehicular Environments (WAVE) – Networking Services, April 2016.

[5] ISO/IEC/IEEE 21450:2010(E). ISO/IEC/IEEE Information technology – Smart transducer interface for sensors and actuators – Common functions, communication protocols, and Transducer Electronic Data Sheet (TEDS) formats, May 2010.

[6] ISO/IEC/IEEE 21451-1:2010(E). ISO/IEC/IEEE Information technology – Smart transducer interface for sensors and actuators – Part 1: Network Capable Application Processor (NCAP) information model, May 2010.

[7] ISO/IEC/IEEE 21451-2:2010(E). ISO/IEC/IEEE Standard for Information technology – Smart transducer interface for sensors and actuators – Part 2: Transducer to microprocessor communication protocols and Transducer Electronic Data Sheet (TEDS) formats, May 2010.

[8] ISO/IEC/IEEE 21451-7:2011(E). Information technology–Smart transducer interface for sensors and actuators–Part 7: Transducers to radio frequency identification (RFID) systems communication protocols and transducer electronic data sheet (TEDS) formats, February 2012.

[9] IEEE Std 754-2008. IEEE Standard for Floating-Point Arithmetic, August 2008.

[10] IEEE Std 802.11-1997. IEEE Standard for Information Technology- Telecommunications and Information Exchange Between Systems-Local and Metropolitan Area Networks-Specific Requirements-Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, November 1997.

[11] IEEE Std 802.11p 2010. IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 6: Wireless Access in Vehicular Environments, July 2010.

[12] IEEE Std 802.15.4-2011. IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs), September 2011.

[13] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, and others. The Design of the Borealis Stream Processing Engine. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, volume 5, pages 277–289, 2005.

[14] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2):120–139, August 2003.

[15] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, C. Erwin, Eduardo F. Galvez, M. Hatoun, Anurag Maskey, Alex Rasin, A. Singer, Michael Stonebraker, Nesime Tatbul, Ying Xing, R. Yan, and Stanley B. Zdonik. Aurora: A Data Stream Management System. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, page 666, 2003.

[16] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. EnviroTrack: towards an environmental computing paradigm for distributed sensor networks. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings*, pages 582–589, 2004.

[17] Karl Aberer, Manfred Hauswirth, and Ali Salehi. The Global Sensor Networks middleware for efficient and flexible deployment and interconnection of sensor networks. Technical Report LSIR-REPORT-2006-006, École polytechnique fédérale de Lausanne, Lausanne, Switzerland, 2006.

[18] Karl Aberer, Manfred Hauswirth, and Ali Salehi. A Middleware for Fast and Flexible Sensor Network Deployment. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 1199–1202, Seoul, Korea, 2006. VLDB Endowment.

[19] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond.* Addison Wesley, Boston, 2005. edition, 2004.

[20] Raman Adaikkalavan and Sharma Chakravarthy. SnoopIB: Interval-Based Event Specification and Detection for Active Databases. In Leonid Kalinichenko, Rainer Manthey, Bernhard Thalheim, and Uwe Wloka, editors, *Advances in Databases and Information Systems*, number 2798 in Lecture Notes in Computer Science, pages 190–204. Springer Berlin Heidelberg, September 2003.

[21] Asaf Adi. Amit — the situation manager. *Int. J. on Very Large Data Bases*, 13:2004, 2004.

[22] Sarfraz Alam, Mohammad MR Chowdhury, and Josef Noll. Senaas: An event-driven sensor virtualization approach for internet of things cloud. In *Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on*, pages 1–6, Suzhou, China, 2010. IEEE.

[23] Hans-Jürgen Appelrath, Dennis Geesen, Marco Grawunder, Timo Michelsen, and Daniela Nicklas. Odysseus: A Highly Customizable Framework for Creating Efficient Event Stream Management Systems. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 367–368, New York, NY, USA, 2012. ACM.

[24] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: The Stanford Stream Data Manager (Demonstration Description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 665–665, New York, NY, USA, 2003. ACM.

[25] Arvind Arasu, Shivnath Babu, and Jennifer Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *Database Programming Languages*, number 2921 in Lecture Notes in Computer Science, pages 1–19. Springer Berlin Heidelberg, September 2003.

[26] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[27] Alfonso Ariza and Alicia Triviño. *Simulation of Multihop Wireless Networks in OMNeT++*, pages 140–158. IGI Global, 2012.

[28] Jean Arlat, Michel Diaz, and Mohamed Kaaniche. Towards resilient cyber-physical systems: The ADREAM project. In *Design & Technology of Integrated Systems In Nanoscale Era (DTIS), 2014 9th IEEE International Conference On*, pages 1–5. IEEE, 2014.

[29] Atmel Cooperation. *IEEE 802.15.4 MAC Software Package - User Guide*, June 2012.

[30] Atmel Corporation. *Low Power,2.4GHz Transceiver for ZigBee, IEEE 802.15.4, 6LoWPAN, RF4CE and ISM Applications*, October 2011.

[31] Ron Ausbrooks, Stephen Buswell, David Carlisle, Giorgi Chavchanidze, Stéphane Dalmas, Stan Devitt, Angel Diaz, Sam Dooley, Roger Hunter, Patrick Ion, Michael Kohlhase, Azzeddine Lazrek, Paul Libbrecht, Bruce Miller, Robert Miner, Chris Rowley, Murray Sargent, Bruce Smith, Neil Soiffer, Robert Sutor, and Stephen Watt. Mathematical Markup Language (MathML) Version 3.0. Technical Report 2, W3C, 2014.

[32] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

[33] Radhakisan Baheti and Helen Gill. Cyber-physical systems. *The impact of control technology*, 12:161–166, 2011.

[34] T. Bangemann, M. Riedl, M. Thron, and C. Diedrich. Integration of Classical Components Into Industrial Cyber Physical Systems. *Proceedings of the IEEE*, 104(5):947–959, May 2016.

[35] Roger S. Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research*, pages 363–374, Pacific Grove, CA, USA, 2007. arXiv: cs/0612115.

[36] Christian Berger, Erik Dahlgren, Johan Grunden, Daniel Gunnarson, Nadia Holtryd, Anmar Khazal, Mohamed Mustafa, Marina Papatriantafilou, Elad Michael Schiller, Christoph Steup, Viktor Swantesson, and Philippas Tsigas. Bridging Physical and Digital Traffic System Simulations with the Gulliver Test-Bed. In *International Workshop on Communication Technologies for Vehicles*, pages 169–184, Lille, France, 2013.

[37] G. Biegel and V. Cahill. A framework for developing mobile, context-aware applications. In *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications, 2004. PerCom 2004*, pages 361 – 365, March 2004.

[38] Bluetooth SIG. *Specification of the Bluetooth System Ver. 4.0*, June 2010.

[39] Malte Brettel, Niklas Friederichsen, Michael Keller, and Marius Rosenberg. How virtualization, decentralization and network building change the manufacturing landscape: An Industry 4.0 Perspective. *International Journal of Mechanical, Aerospace, Industrial, Mechatronic and Manufacturing Engineering*, 8(1):37–44, 2014.

[40] R. Cardell-Oliver, M. Reynolds, and M. Kranz. A space and time requirements logic for sensor networks. In *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*, pages 283–289, Nov 2006.

[41] Donald Carney, Ugur \cC}etintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 215–226, 2002.

[42] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, August 2001.

[43] Sharma Chakravarthy and Deepak Mishra. Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering*, 14(1):1–26, 1994.

[44] Guanling Chen and David Kotz. Policy-driven data dissemination for context-aware applications. In *In Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*, page 283–289, 2005.

[45] Guanling Chen, Ming Li, and David Kotz. Design and implementation of a large-scale context fusion network. In *First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (Mobiquitous)*, page 246–255, 2004.

[46] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *In CIDR*, 2003.

[47] Thomas M. Connolly and Carolyn E. Begg. Chapter 21 Query Processing. In *Database systems: a practical approach to design, implementation, and management*, pages 630–664. Pearson Education, 2. edition, 2005.

[48] Thomas M. Connolly and Carolyn E. Begg. Chapter 4 Relational Algebra and Relational Calculus. In *Database systems: a practical approach to design, implementation, and management*, pages 88–110. Pearson Education, 2. edition, 2005.

[49] AutomationML Consortium. Whitepaper AutomationML Communication, September 2014.

[50] AutomationML Consortium. Whitepaper AutomationML Part 1 - Architecture and general requirements, April 2016.

[51] AutomationML Consortium. Whitepaper AutomationML Part 4: AutomationML Logic, January 2017.

[52] Pedro Costa, José Ricardo Parizzi, Rolf Johansson, Elad Michael Schiller, Oscar Morales, Renato Librino, António Casimiro, Joerg Kaiser, and Siavash Aslani. First Report on the KARYON Architecture. Technical Report D2.1, European Commission 7th Framework Program - ICT, April 2012.

[53] Flaviu Cristian. Probabilistic clock synchronisation. *Distributed Computing*, 3:146–158, September 1989.

[54] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen. Generative Programming and Active Libraries. In M. Jazayeri, D. Musser, and R. Loos, editors, *Proceedings of Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 25–39. Springer-Verlag, 2000.

[55] Hend Dawood. Chapter 1 Prologue: A Weapon Against Uncertainty. In *Theories of Interval Arithmetic: Mathematical Foundations and Applications*, pages 1–6. LAP Lambert Academic Publishing, October 2011.

[56] Hend Dawood. Chapter 2 The classical Theory of Interval Arithmetic. In *Theories of Interval Arithmetic: Mathematical Foundations and Applications*, pages 7–24. LAP Lambert Academic Publishing, October 2011.

[57] Arnaldo Carvalho de Melo. The New Linux 'perf' tools. In *Slides from Linux Kongress*, volume 18, Nuremberg, Germany, September 2010.

[58] dresden electronik. deRFsam3-23M10. `http://www.dresden-elektronik.de/funktechnik/products/radio-modules/oem-modules-derfsam3`. accessed 13. Aug. 2017.

[59] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.

[60] E. Einhorn, T. Langner, R. Stricker, C. Martin, and H. M. Gross. MIRA - middleware for robotic applications. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2591–2598, Vilamoura, Portugal, October 2012. IEEE.

[61] Wilfried Elmenreich. An Introduction to Sensor Fusion. Research Report 47/2001, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2001.

[62] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronisation using reference broadcasts. In *SIGOPS*, volume 36, pages 147–163, New York, NY, USA, December 2002. ACM.

[63] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[64] Adrian Fitzpatrick, Gregory Biegel, Siobhán Clarke, and Vinny Cahill. Towards a sentient object model. In *Workshop on Engineering Context-Aware Object Oriented Systems and Environments (ECOOSE)*. Citeseer, 2002.

[65] T. Foote. tf: The transform library. In *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*, pages 1–6, April 2013.

[66] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17 – 37, 1982.

[67] 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*, DEBS '12, pages 179–190, New York, NY, USA, 2012. ACM.

[68] Ernest Friedman-Hill and others. Jess, the rule engine for the java platform. Technical Report 7.2p2, Sandia National Laboratories, 2008.

[69] Luotao Fu and Robert Schwebel. RT PREEMPT HOWTO. `https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO`. accessed 13. Aug. 2017.

[70] Richard Helm Gamma, Ralph Johnson & John Vlissidess Erich. *Design Patterns: Elements Of Reusable Object-Oriented Software*. Pearson Education, 1994.

[71] Damien D. George, Paul Sokolovsky, and others. The micropython language. `http://micropython.org`. accessed 13. Aug. 2017.

[72] Martin Gergeleit and Hermann Streich. Implementing a distributed high-resolution real-time clock using the CAN-bus. In *Proceedings of the 1st international CAN-Conference*, Mainz, Germany, 1994. Can-in-Automation (CIA).

[73] James Gosling, Bill Joy, Guy L Steele, Gilad Bracha, and Alex Buckley. *The Java language specification.* Pearson Education, 2014.

[74] Gaël Guennebaud. Eigen: a c++ linear algebra library. `http://downloads.tuxfamily.org/eigen/eigen_aristote_may_2013.pdf`, May 2013. accessed 13. Aug. 2017.

[75] Gaël Guennebaud, Benoît Jacob, and others. Eigen v3. `http://eigen.tuxfamily.org`, 2010. accessed 13. Aug. 2017.

[76] Aleksey Gurtovoy and David Abrahams. THE BOOST MPL LIBRARY. `http://www.boost.org/doc/libs/1_64_0/libs/mpl/doc/index.html`, November 2004. accessed 13. Aug. 2017.

[77] John W Hager, James F Behensky, and Brad W Drew. The universal grids: Universal Transverse Mercator (UTM) and Universal Polar Stereographic (UPS). *DMA technical manual*, 8358.2, 1989.

[78] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible Network Experiments Using Container-based Emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 253–264, New York, NY, USA, 2012. ACM.

[79] Godfrey Harold Hardy, EM Wright, Roger Heath-Brown, and Joseph Silverman. Statement of the fundamental theorem of arithmetics. In *An Introduction to the Theory of Numbers*, pages 3–4. Oxford University Press, Oxford, 6th edition, 2008.

[80] Vincent Huang and Muhammad Kashif Javed. Semantic sensor information description and processing. In *Sensor Technologies and Applications, 2008. SENSORCOMM'08. Second International Conference on*, pages 456–461, Cap Esterel, France, 2008. IEEE.

[81] Daniel Hughes, Klaas Thoelen, Wouter Horré, Nelson Matthys, Javier Del Cid, Sam Michiels, Christophe Huygens, Wouter Joosen, and Jo Ueyama. Building Wireless Sensor Network Applications with LooCI. In *Advancing the Next-Generation of Mobile Computing: Emerging Technologies: Emerging Technologies*, pages 61–85. IGI Global, Hershey, PA, USA, 1 edition, February 2012.

[82] International Bureau of Weights and Measures. The International System of Units (SI), 2006.

[83] Jörg Kaiser, José Rufino, Christoph Steup, Tino Brade, José Rufino, Jeferson Souza, Rui Caldeira, and André Guerreiro. Working prototype of adaptive middleware. Technical Report D3.3, European Commission 7th Framework Program - ICT, December 2013.

[84] Elliott Kaplan. *Understanding GPS Principles and Applications.* Artech House Mobile Communicat, Boston, February 1996.

[85] Mumraiz Khan Kasi, Annika Hinze, Catherine Legg, and Steve Jones. SEPSen: Semantic Event Processing at the Sensor Nodes for Energy Efficient Wireless Sensor Networks. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 119–122, New York, NY, USA, 2012. ACM.

[86] Eli Katsiri, Jean Bacon, and Alan Mycoft. An extended Publish/Subscribe protocol for transparent subscriptions to distributed abstract state in sensor driven systems using abstract events. In *Proceedings of the International Workshop on Distributed Event-based Systems (DEBS)*, pages 68–73, Edinburgh, Great Britain, May 2004.

[87] Robert Katz. 2-2 The Lorentz Transformation: Simultaneity and Time Sequence. In *An Introdcution to the Special Theory of Relativity*, pages 31–33. D. Van Nostrand Company, 1964.

[88] V. Kodaganallur. Incorporating language processing into Java applications: a JavaCC tutorial. *IEEE Software*, 21(4):70–77, July 2004.

[89] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer Science+Business Media, 2nd edition, 2011.

[90] A. Köpke, M. Swigulski, K. Wessel, D. Willkomm, P. T. Klein Haneveld, T. E. V. Parker, O. W. Visser, H. S. Lichte, and S. Valentin. Simulating Wireless and Mobile Networks in OMNeT++ the MiXiM Vision. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, page 71:1–71:8, Brussels, Belgium, March 2008. Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering.

[91] Vladimir Kostyukov. la4j. `http://la4j.org`. accessed 31. Oct. 2017.

[92] Mark Kranz. *SENSID: a Situation Detector for Sensor Networks*. PhD thesis, University of Western Australia, June 2005.

[93] Rudolf Kruse, Christian Borgelt, Christian Braune, Sanaz Mostaghim, Matthias Steinbrecher, Frank Klawonn, and Christian Moewes. Bayes and Markov Networks. In *Computational Intelligence: A Methodological Introduction*, pages 457–563. Springer, New York, NY, 2nd ed. 2016 edition, September 2016.

[94] Christian Kuka, Sebastian Gerwinn, Sören Schweigert, Sönke Eilers, and Daniela Nicklas. Context-model Generation for Safe Autonomous Transport Vehicles. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 365–366, New York, NY, USA, 2012. ACM.

[95] Christian Kuka and Daniela Nicklas. Quality Matters: Supporting Quality-aware Pervasive Applications by Probabilistic Data Stream Management. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 1–12, New York, NY, USA, 2014. ACM.

[96] Chris Lattner. LLVM – The LLVM Compiler Infrastructure. `http://llvm.org/`. accessed 13. Aug. 2017.

[97] Chris Arthur Lattner. *LLVM: An infrastructure for multi-stage optimization.* PhD thesis, University of Illinois at Urbana-Champaign, 2002.

[98] Edward A. Lee. Cyber Physical Systems: Design Challenges. Technical Report UCB/EECS-2008-8, EECS Department, University of California, Berkeley, January 2008.

[99] Jay Lee, Behrad Bagheri, and Hung-An Kao. Recent advances and trends of cyber-physical systems and big data analytics in industrial informatics. In *Proceedings of the 12th International Conference on Industrial Informatics (INDIN)*, Porto Alegre, Brazil, 2014. IEEE.

[100] Jay Lee, Behrad Bagheri, and Hung-An Kao. A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems. *Manufacturing Letters*, 3:18–23, January 2015.

[101] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An Operating System for Sensor Networks. In Werner Weber, Jan M. Rabaey, and Emile Aarts, editors, *Ambient Intelligence*, pages 115–148. Springer Berlin Heidelberg, 2005.

[102] Guoli Li and Hans-Arno Jacobsen. Composite Subscriptions in Content-Based Publish/Subscribe Systems. In Gustavo Alonso, editor, *Middleware 2005*, number 3790 in Lecture Notes in Computer Science, pages 249–269. Springer Berlin Heidelberg, November 2005.

[103] Ray Lischner. *STL Pocket Reference.* O'Reilly & Associates, Sebastopol, CA, USA, 1 edition, 2003.

[104] Samuel Madden, Michael J Franklin, Joseph Hellerstein, and Wei Hong. TAG: a Tiny Aggregation Service for Ad-Hoc Sensor Networks. In *Proceedings of the 5th symposium on Operating systems design and implementation*, volume 36, pages 131–146, New York, NY, USA, 2002. ACM.

[105] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems (TODS) - Special Issue: SIGMOD/PODS 2003*, 30(1):122–173, 2005.

[106] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige. The Office Marathon: Robust navigation in an indoor office environment. In *2010 IEEE International Conference on Robotics and Automation*, pages 300–307. IEEE, May 2010.

[107] Pedro José Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel. FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks. In *Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN)*, pages 212–227, Zurich, Switzerland, 2006. Springer.

[108] Keith Marzullo. Implementing fault-tolerant sensors. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, Santa Monica, Ca, USA, December 1989. IEEE Computer Society Press.

[109] Friedemann Mattern and Christian Floerkemeier. From the Internet of Computers to the Internet of Things. In *From Active Data Management to Event-Based Systems and More*, Lecture Notes in Computer Science, pages 242–259. Springer, Berlin, Heidelberg, 2010.

[110] Brian McBride. Jena: Implementing the RDF Model and Syntax Specification. In *Proceedings of the Second International Conference on Semantic Web - Volume 40*, SemWeb'01, pages 23–28, Aachen, Germany, 2001.

[111] Geoffrey McLachlan and David Peel. Normal Scale Mixture Model. In *Finite Micture Models*, page 17. John Wiley & Sons, Ltd., 2001.

[112] M. Mock, R. Frings, E. Nett, and S. Trikaliotis. Continuous clock synchronisation in wireless real-time applications. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, pages 125–132, October 2000.

[113] Abdul-Wahid Mohammed, Yang Xu, Ming Liu, and Haixiao Hu. Semantical Markov Logic Network for Distributed Reasoning in Cyber-Physical Systems. *Journal of Sensors*, 2017(4259652):1–15, 2017.

[114] Lucas Nussbaum and Olivier Richard. A Comparative Study of Network Link Emulators. In *Proceedings of the 2009 Spring Simulation Multiconference*, SpringSim '09, pages 85:1–85:8, San Diego, CA, USA, 2009. Society for Computer Simulation International.

[115] Dan O'Keeffe. Distributed complex event detection for pervasive computing. Technical Report UCAM-CL-TR-783, University of Cambridge, Computer Laboratory, July 2010.

[116] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[117] Open Geospatial Consortium 12-000. *OGC® SensorML: Model and XML Encoding Standard*, February 2014.

[118] Angela Orebaugh, Gilbert Ramirez, and Jay Beale. Chapter 4 Using Wireshark - Analyze. In *Wireshark & Ethereal network protocol analyzer toolkit*, pages 178–189. Syngress, 2006.

[119] Santashil PalChaudhuri, Amit Saha, and David B Johnson. Probabilistic clock synchronisation service in sensor networks. *IEEE Transactions on Networking*, 2:177–189, 2003.

[120] Michael Paleczny, Christopher Vick, and Cliff Click. The Java hotspotTM Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[121] C. Perkins, E. Belder-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561, RFC Editor, July 2003.

[122] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE Proceedings of the 22nd International Conference on Data Engineering*, pages 49–61, Atlanta, GA, USA, 2006.

[123] Peter R. Pietzuch. Hermes: A scalable event-based middleware. Technical Report 590, University of Cambridge, Computer Laboratory, 2004.

[124] Su Ping. Delay measurement time synchronisation for wireless sensor networks. Technical Report IRB-TR-03-013, Intel Research Berkeley Lab, 2003.

[125] György Pongor. OMNeT: objective modular network testbed. In *Proceedings of the International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, page 323–326, San Diego, CA, USA, October 1993. Society for Computer Simulation International.

[126] D. Preuveneers and Y. Berbers. Encoding Semantic Awareness in Resource-Constrained Devices. *IEEE Intelligent Systems*, 23(2):26–33, March 2008.

[127] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. In Isabel Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Mike Uschold, and Lora M. Aroyo, editors, *The Semantic Web - ISWC 2006*, number 4273 in Lecture Notes in Computer Science, pages 30–43. Springer Berlin Heidelberg, November 2006. DOI: 10.1007/11926078_3.

[128] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, volume 3, page 5, 2009.

[129] Raspberry Pi Foundation. Rapsberry Pi Model A+. `http://www.raspberrypi.org/products/model-a-plus/`. accessed 13. Aug. 2017.

[130] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-addressable Network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 161–172, New York, NY, USA, 2001. ACM.

[131] Marc Recksiedler. Master thesis: Synergieeffekte von GeoCasting in einem drahtlosen, Ad-hoc Publish/Subscribe-System, 2013.

[132] Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, February 2006.

[133] E. Rohmer, S. P. N. Singh, and M. Freese. V-REP: A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1321–1326, November 2013.

[134] K. Römer and F. Mattern. The design space of wireless sensor networks. *IEEE Wireless Communications*, 11(6):54–61, December 2004.

[135] Kay Römer. Time synchronisation in ad hoc networks. In *Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, MobiHoc '01, pages 173–182, New York, NY, USA, 2001. ACM.

[136] Kay Römer and Friedemann Mattern. Event-based systems for detecting real-world states with sensor networks: A critical analysis. In *Proceedings of the Intelligent Sensors, Sensor Networks and Information Processing Conference*, pages 389–396. IEEE, 2004.

[137] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, page 329–350, London, UK, UK, 2001. Springer-Verlag.

[138] Jedrzej Rybicki, Björn Scheuermann, Markus Koegel, and Martin Mauve. PeerTIS: a peer-to-peer traffic information system. In *Proceedings of the sixth ACM international workshop on VehiculAr InterNETworking*, VANET '09, pages 23–32, New York, NY, USA, September 2009. ACM.

[139] Christopher M. Sadler and Margaret Martonosi. Data Compression Algorithms for Energy-constrained Devices in Delay Tolerant Networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 265–278, New York, NY, USA, 2006. ACM.

[140] Matthias C. Schabel and Steven Watanabe. Chapter 43. Boost.Units 1.1.0. `http://www.boost.org/doc/libs/1_64_0/doc/html/boost_units.html`, April 2017. accessed 13. Aug. 2017.

[141] Michael Schiefer, Christoph Steup, and Jörg Kaiser. Real World Testing of Aggregation in Publish/Subscribe Systems. In *Proceedings of Wireless Information Networks and Systems (WINSYS)*, pages 1–8, Reykjavik, Iceland, August 2013. IEEE.

[142] Michael Schulze. *Adaptierbare ereignisbasierte Middleware für ressourcenbeschränkte Systeme.* PhD thesis, Otto-von-Guericke-University, Magdeburg, Germany, 2011.

[143] Boris Schäling. Part IX. Functional Programming. In *The Boost C++ Libraries.* XML Press, Laguna Hills, Calif, 2. vollständ. überarb. edition, September 2014.

[144] Boris Schäling. Part XV. Application Libraries. In *The Boost C++ Libraries.* XML Press, Laguna Hills, Calif, 2. vollständ. überarb. edition, September 2014.

[145] Amit P. Sheth. *Interoperating Geographic Information Systems*, chapter Changing Focus on Interoperability in Information Systems: From System, Syntax, Structure to Semantics, pages 5–29. Springer US, Boston, MA, USA, 1999.

[146] J. Shi, J. Wan, H. Yan, and H. Suo. A survey of Cyber-Physical Systems. In *2011 International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 1–6, November 2011.

[147] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual.* Addison-Wesley Professional, 1 edition, 2001.

[148] Doug Simon and Cristina Cifuentes. The squawk virtual machine: Java(TM) on the bare metal. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 150–151, New York, NY, USA, 2005. ACM.

[149] Thirunavukkarasu Sivaharan, Gordon S. Blair, Adrian Friday, Maomao Wu, Hector Duran-Limon, Paul Okanda, and Carl-Fredrik Sørensen. Cooperating sentient vehicles for next generation automobiles. In *ACM/USENIX MobiSys 2004 International Workshop on Applications of Mobile Embedded Systems (WAMES 2004 online proceedings)*, Boston, MA, USA, June 2004.

[150] Bruce Snyder, Dejan Bosanac, and Rob Davies. *ActiveMQ in Action*. Manning Publications Co., Greenwich, CT, USA, 2011.

[151] Dirk Steindorf. Master Thesis: Extended Robot Navigation Using Dynamically Generated Occupancy Grids, 2016.

[152] Christoph Steup. Collaborative Sensing in Wireless Sensor Actor Networks, February 2014. Presentation: Doktorandentag of the Computer Science Faculty of the Otto-von-Guericke University Magdeburg.

[153] Christoph Steup. ASEIA - ROS. `https://github.com/steup/ASEIA-ROS`, January 2015. accessed at 2017-11-07.

[154] Christoph Steup, Michael Schulze, and Jörg Kaiser. Exploiting Template-Metaprogramming for Highly Adaptable Device Drivers a Case Study on CANARY an AVR CAN-Driver. In *Proceedings of 12th Brazilian Workshop on Real-Time and Embedded Systems (WTR)*, Gramado, Brazil, 2010. Brazilian Computer Society.

[155] Christoph Steup, Sebastian Zug, and Jörg Kaiser. Evaluation of an Uncertainty Aware Hybrid Clock Synchronisation System for Wireless Sensor Networks. *International Journal on Advances in Networks and Services*, 8(1&2):54–68, 2015.

[156] Christoph Steup, Sebastian Zug, Jörg Kaiser, and Andy Bruehan. Uncertainty Aware Hybrid Clock Synchronisation in Wireless Sensor Networks. In *Proceedings of Eighth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM)*. IARIA, July 2014.

[157] Christoph Steup, Sebastian Zug, and Jörg Kasier. Achieving Cooperative Sensing in Automotive Scenarios through Complex Event Processing. In *Proceedings of Seventh International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM)*, Porto, Portugal, July 2013. IARIA.

[158] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4 edition, July 2013.

[159] Xiaoyu Tong and E.C. Ngai. A Ubiquitous Publish/Subscribe Platform for Wireless Sensor Networks with Mobile Mules. In *2012 IEEE 8th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 99–108, May 2012.

[160] A. Vahidi and A. Eskandarian. Research advances in intelligent collision avoidance and adaptive cruise control. *IEEE Transactions on Intelligent Transportation Systems*, 4(3):143 – 153, September 2003.

[161] Guido van Rossum and Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011.

[162] VAST Team. sensiasoft/lib-sensorml. `https://github.com/sensiasoft/lib-sensorml`. accessed 13. Aug. 2017.

[163] Todd L. Veldhuizen. C++ Templates as Partial Evaluation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Tech. Report NS-99-1, pages 13–18. BRICS, 1999.

[164] Todd L Veldhuizen. Just when you thought your little language was safe:“expression templates” in Java. In *International Symposium on Generative and Component-Based Software Engineering*, pages 189–201. Springer, 2000.

[165] P. Verissimo and A. Casimiro. Event-driven support of real-time sentient objects. In *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems, 2003. (WORDS 2003)*, pages 2–9, January 2003.

[166] Paulo Verissimo, Luis Rodrigues, and Antonio Casimiro. Cesiumspray: a precise and accurate global time service for large-scale systems. *Real-Time Systems*, 12(3):243–294, 1997.

[167] Holger Wache, Thomas Voegele, Ubbo Visser, Heiner Stuckenschmidt, Gerhard Schuster, Holger Neumann, and Sebastian Hübner. Ontology-based integration of information-a survey of existing approaches. In *Proceedings of IJCAI-01 Workshop: Ontologies and Information Sharing*, volume 2001, pages 108–117, Seattle, USA, 2001.

[168] Kamin Whitehouse, Feng Zhao, and Jie Liu. Semantic Streams: A Framework for Composable Semantic Interpretation of Sensor Data. In *Wireless Sensor Networks*, Lecture Notes in Computer Science, pages 5–20. Springer, Berlin, Heidelberg, February 2006.

[169] Liyang Yu, Neng Wang, and Xiaoqiao Meng. Real-time forest fire detection with wireless sensor networks. In *2005 International Conference on Wireless Communications, Networking and Mobile Computing, 2005. Proceedings*, volume 2, pages 1214–1217, September 2005.

[170] Sebastian Zug, André Dietrich, Marc Schappeit, Christoph Steup, and Jörg Kaiser. Flexible Daten-Akquisition & Interpretation für verteilte Sensor-Aktor-Systeme im Produktionsumfeld. In *Forschung und Innovation: 10. Magdeburger Maschienenbautage*, Magdeburg, Germany, 2011.

[171] Sebastian Zug, André Dietrich, Christoph Steup, Tino Brade, and Thomas Petig. Phase optimization for control/fusion applications in dynamically composed sensor networks. In *In Proceedings of the 2013 IEEE International Symposium on Robotic and Sensors Environments (ROSE)*, Washington, DC, USA, 2013. IEEE Computer Society.

[172] Sebastian E. Zug. *Architektur für verteilte, fehlertolerante Sensor-Aktor-Systeme / von Sebastian Ernst Zug*. PhD thesis, Otto-von-Guericke University, 2011.

A. Additional Evaluation Results of the Automotive Scenario

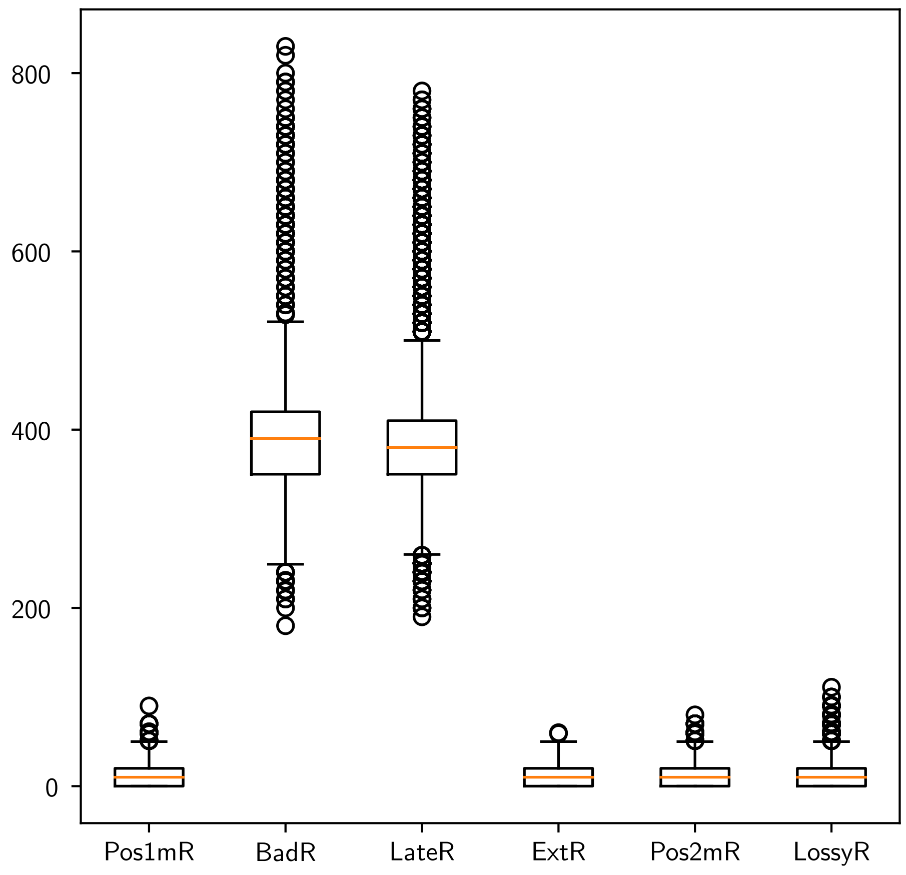
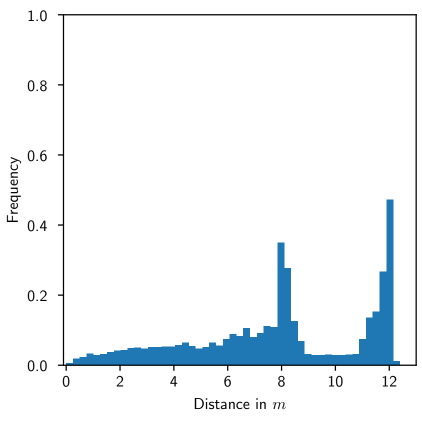
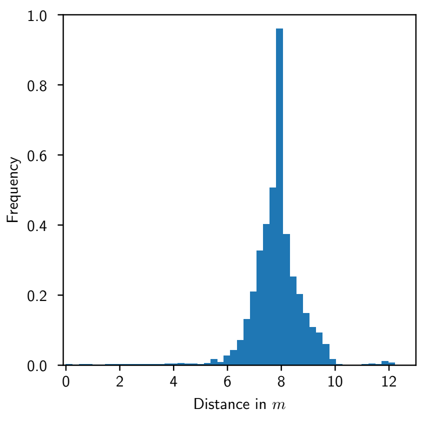


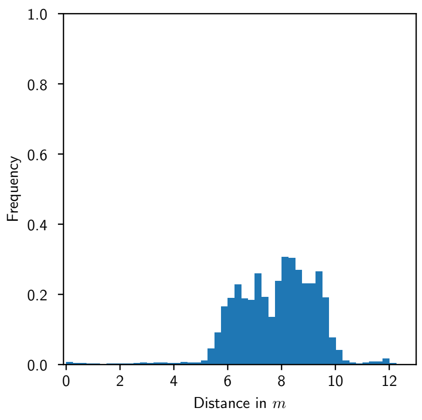
Figure A.1.: Box-Whisker plot showing the delay induced by the in-network processing for the *Road* test cases. The delay is measured from atomic event generation until reception of the resulting complex event.



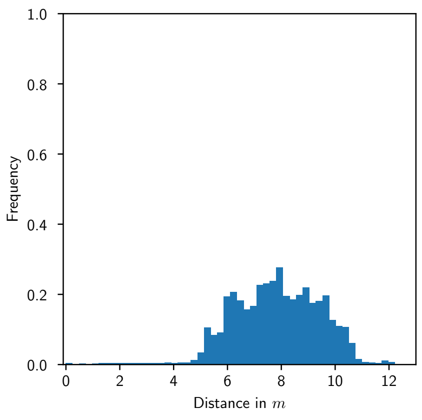
(a) Experiment **Basic**.



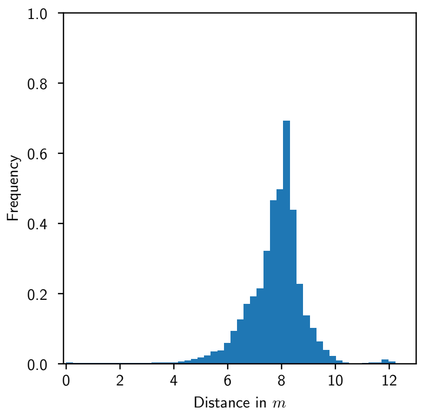
(b) Experiment **Ext**.



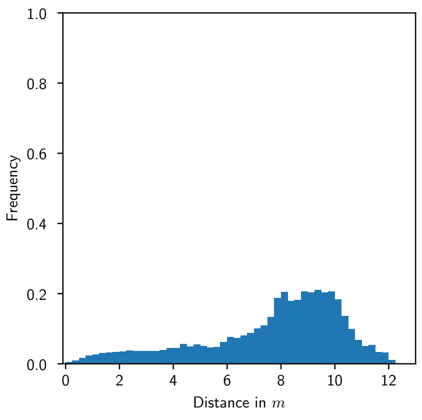
(c) Experiment **Pos1**.



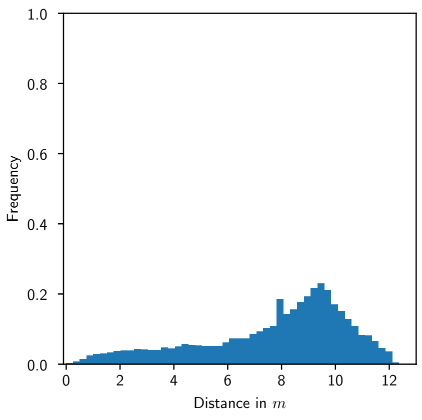
(d) Experiment **Pos2**.



(e) Experiment **Lossy**.

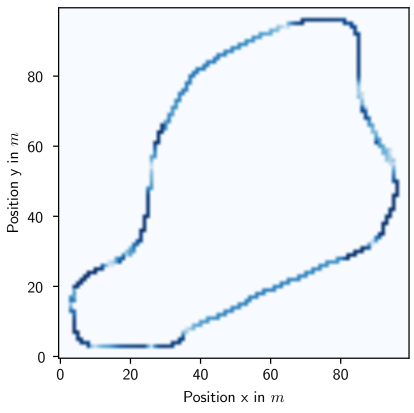


(f) Experiment **Late**.

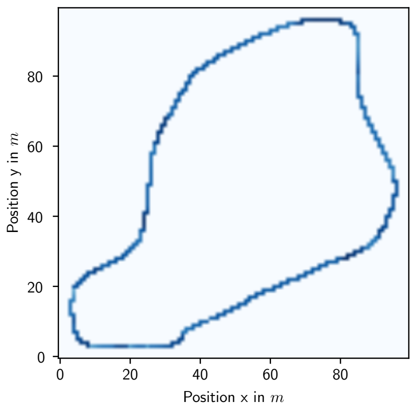


(g) Experiment **Bad**.

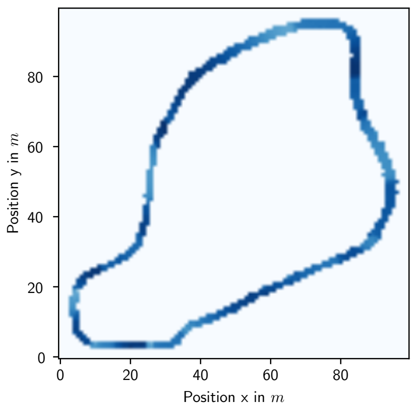
Figure A.2.: Comparison of the different evaluation scenarios regarding **Speed** of the first car as histograms for the *UTM Transformation Graph*.



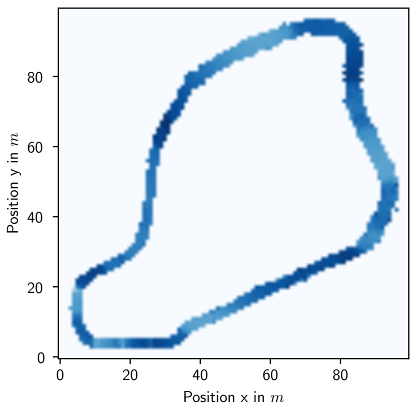
(a) Experiment **Basic**.



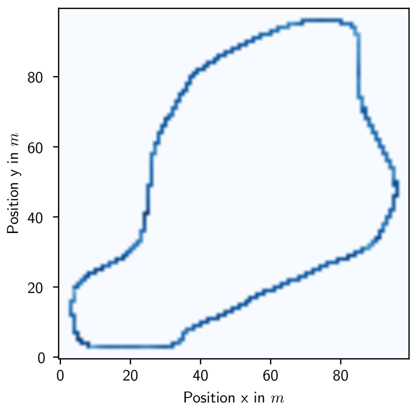
(b) Experiment **Ext**.



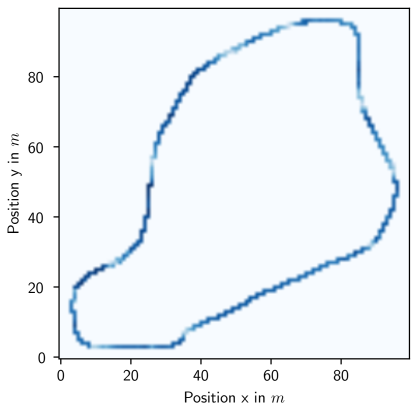
(c) Experiment **Pos1**.



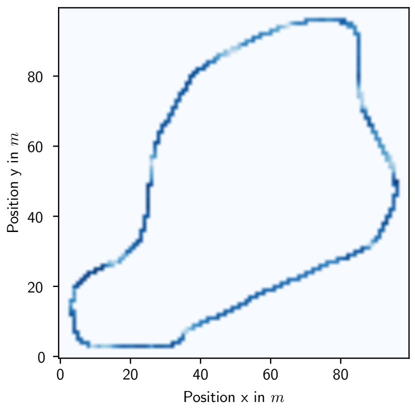
(d) Experiment **Pos2**.



(e) Experiment **Lossy**.

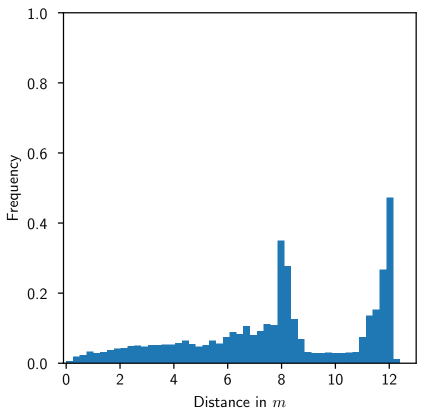


(f) Experiment **Late**.

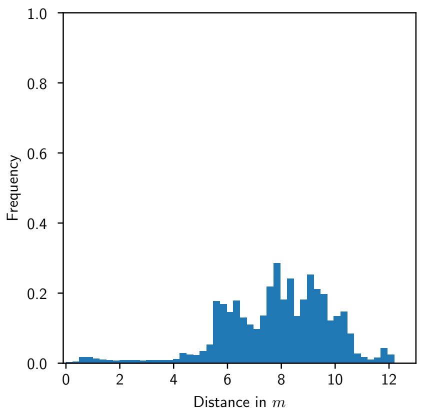


(g) Experiment **Bad**.

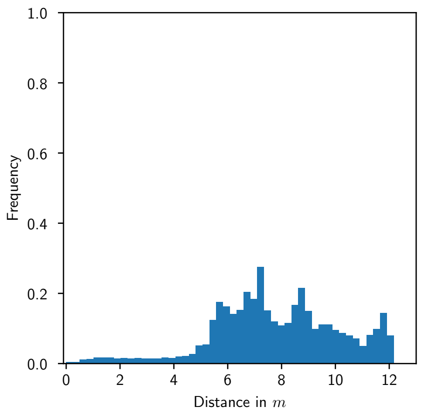
Figure A.3.: Weighted heat map of the absolute speed of car 0 as result of the *UTM Transformation Graph*. White areas indicate zero speed, dark blue areas indicate maximum speed of 12 m/s.



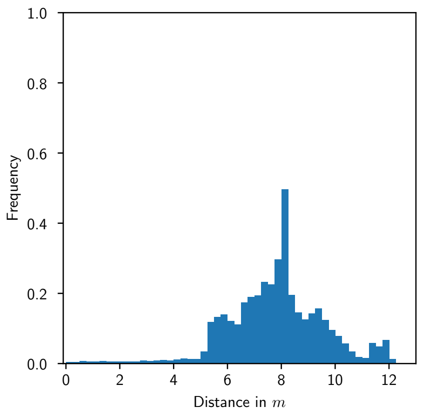
(a) Experiment **Basic**.



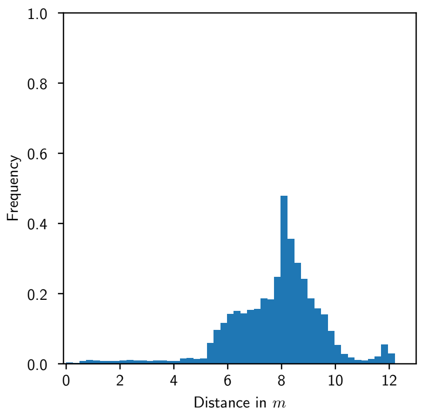
(b) Experiment **ExtR**.



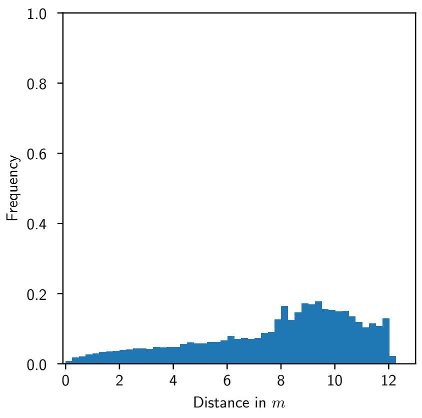
(c) Experiment **Pos1R**.



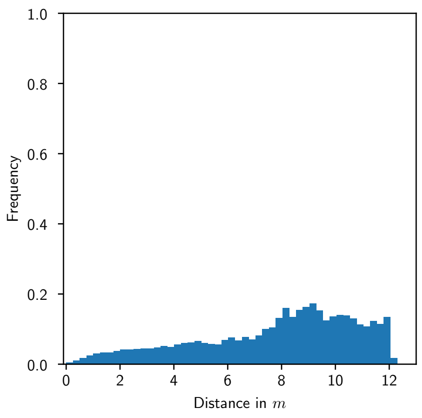
(d) Experiment **Pos2R**.



(e) Experiment **LossyR**.

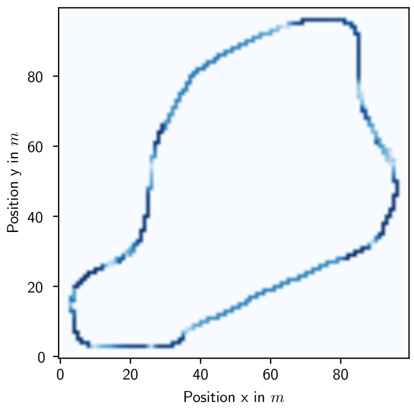


(f) Experiment **LateR**.

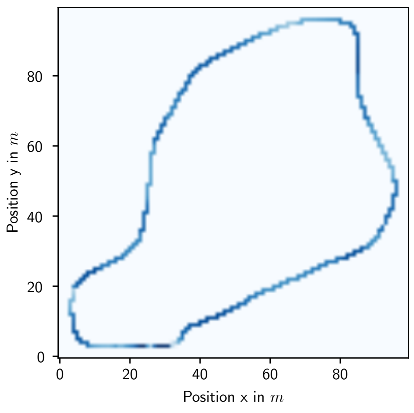


(g) Experiment **BadR**.

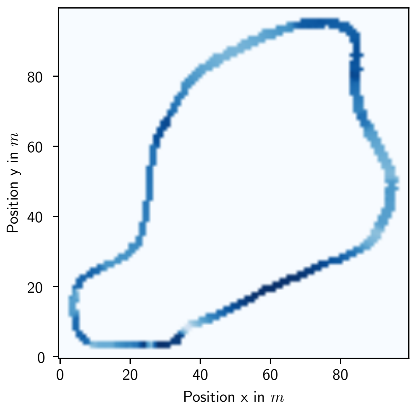
Figure A.4.: Comp of the different evaluation scenarios regarding **Speed** of the first car as histograms for the *Road Transformation Graph*.



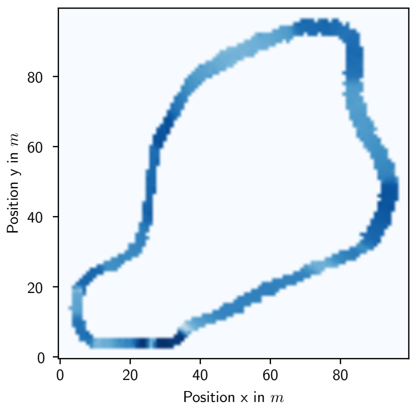
(a) Experiment **Basic**.



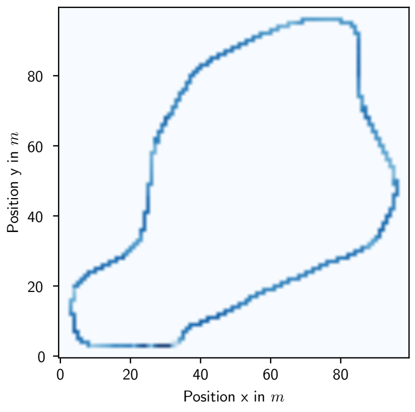
(b) Experiment **Ext**.



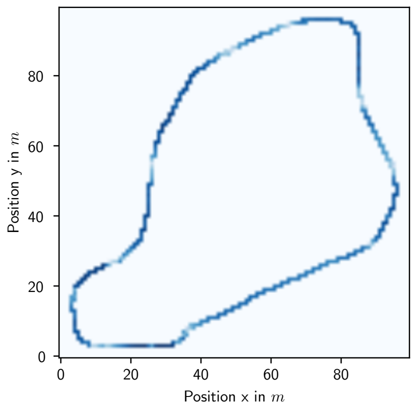
(c) Experiment **Pos1R**.



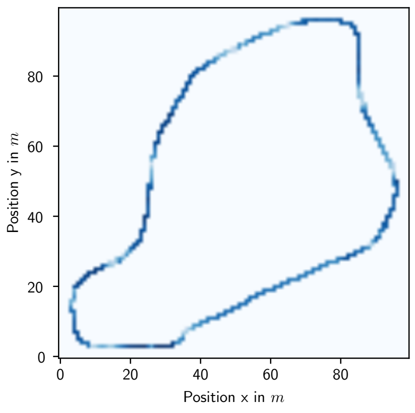
(d) Experiment **Pos2**.



(e) Experiment **LossyR**.

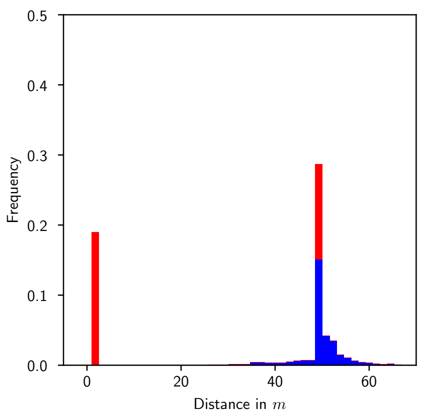


(f) Experiment **LateR**.

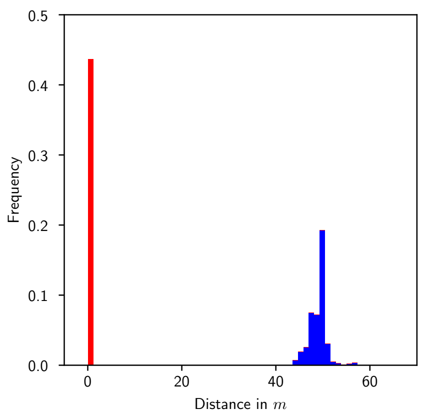


(g) Experiment **BadR**.

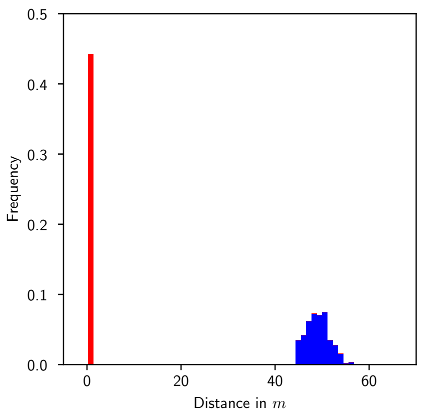
Figure A.5.: Weighted heat map of the absolute speed of car 0 as result of the *Road Transformation Graph*. White areas indicate zero speed, dark blue areas indicate maximum speed of 12 m/s.



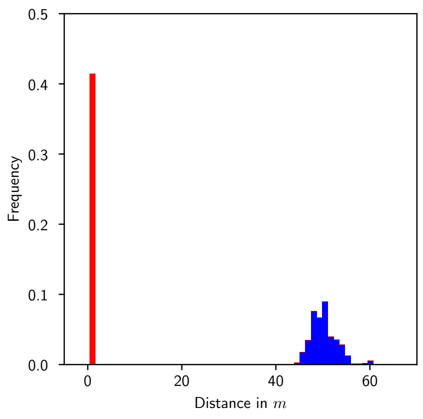
(a) Experiment **Basic**.



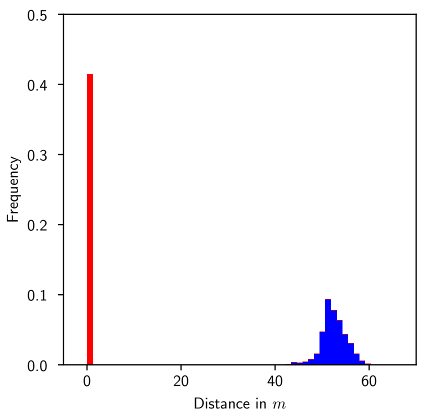
(b) Experiment **Ext**.



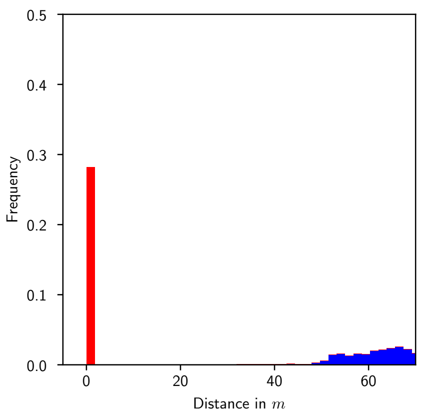
(c) Experiment **Pos1**.



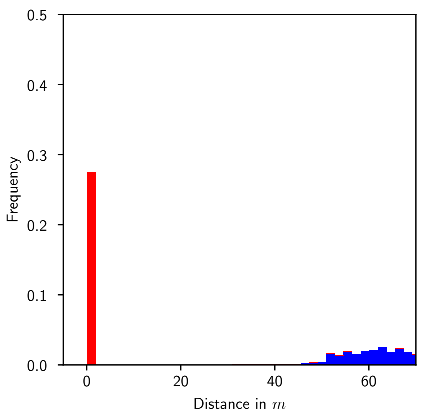
(d) Experiment **Pos2**.



(e) Experiment **Lossy**.

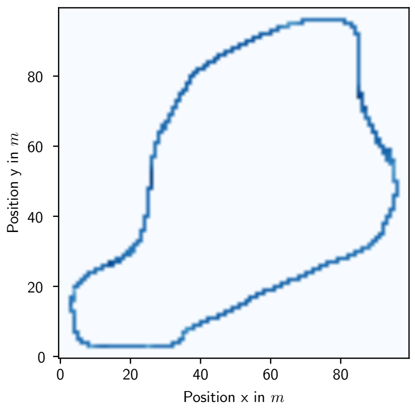


(f) Experiment **Late**.

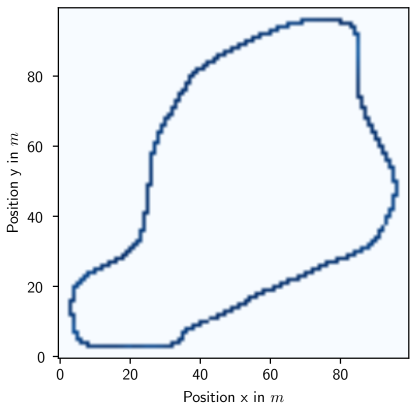


(g) Experiment **Bad**.

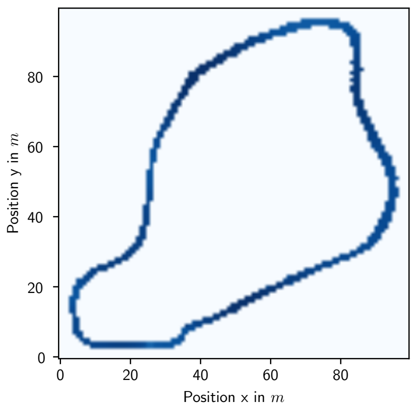
Figure A.6.: Comparison of the different road evaluation scenarios regarding **Distance** and **Distance** uncertainty of the first car as histograms produced by the *UTM extended ACC*. Blue Bars indicate the **Distance** value and red bars indicate **Distance** uncertainty.



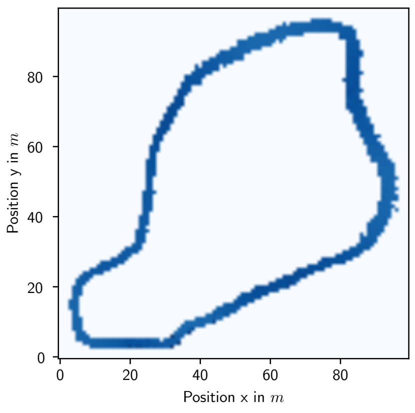
(a) Experiment **Basic**.



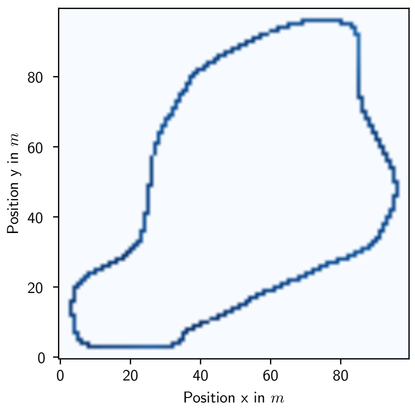
(b) Experiment **Ext**.



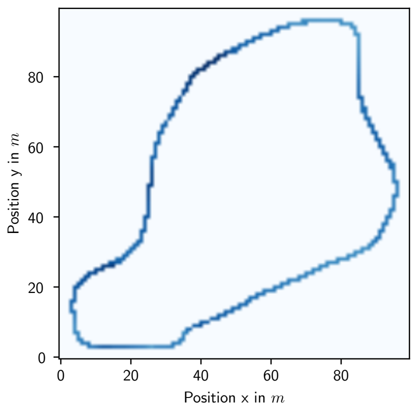
(c) Experiment **Pos1**.



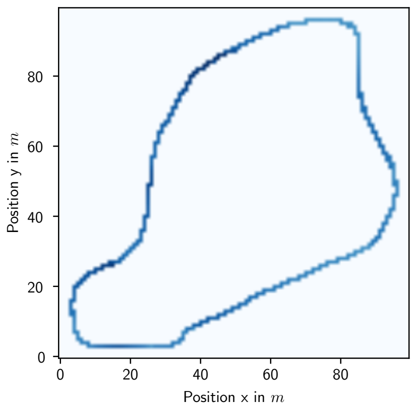
(d) Experiment **Pos2**.



(e) Experiment **Lossy**.

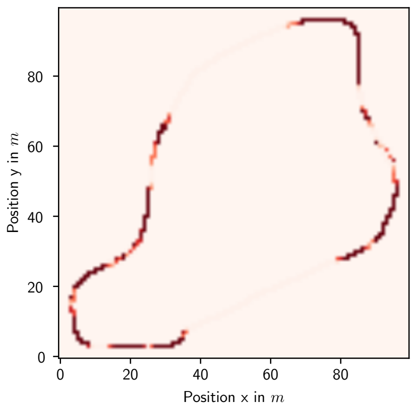


(f) Experiment **Late**.

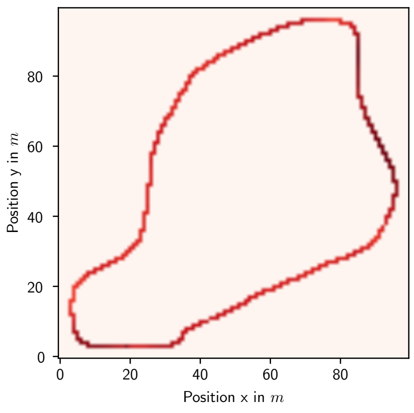


(g) Experiment **Bad**.

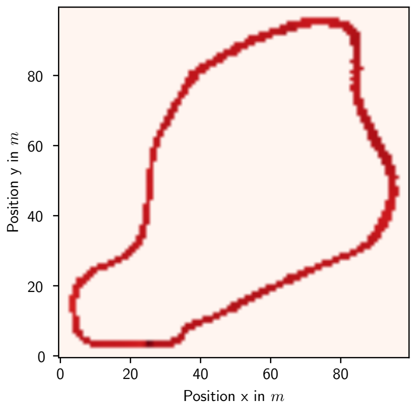
Figure A.7.: Weighted heat map of the absolute distance of car 0 produced by the *UTM extended ACC*. White areas indicate zero distance, dark blue areas indicate maximum distance of 100 m.



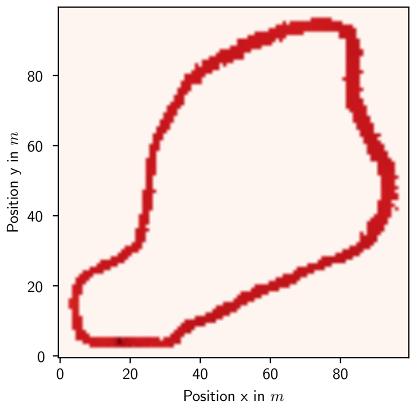
(a) Experiment **Basic**.



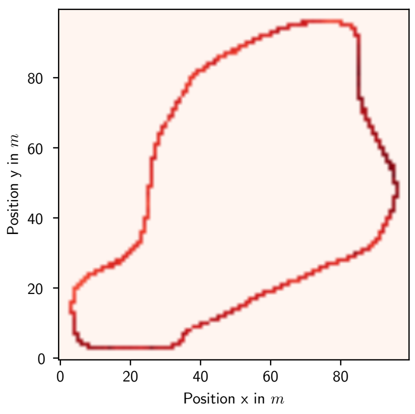
(b) Experiment **Ext**.



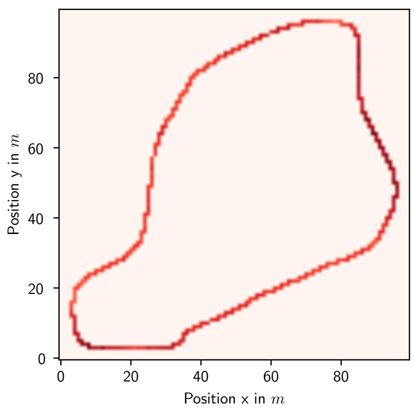
(c) Experiment **Pos1**.



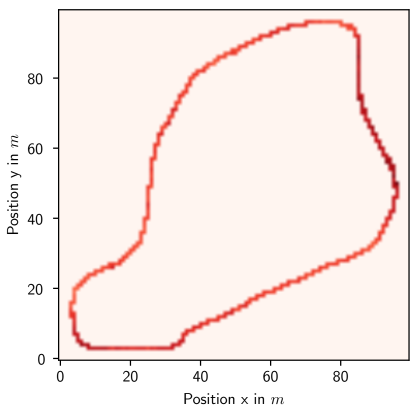
(d) Experiment **Pos2**.



(e) Experiment **Lossy**.

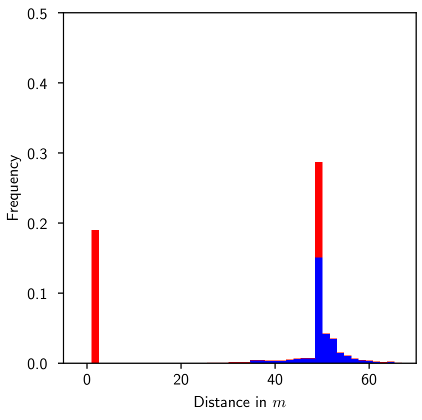


(f) Experiment **Late**.

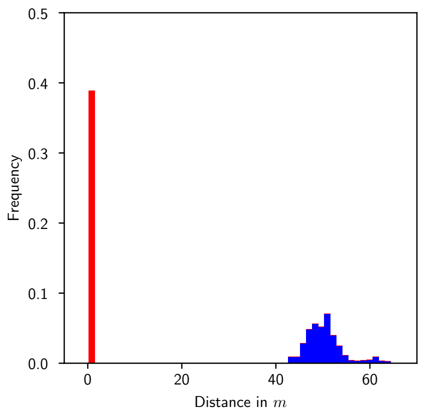


(g) Experiment **Bad**.

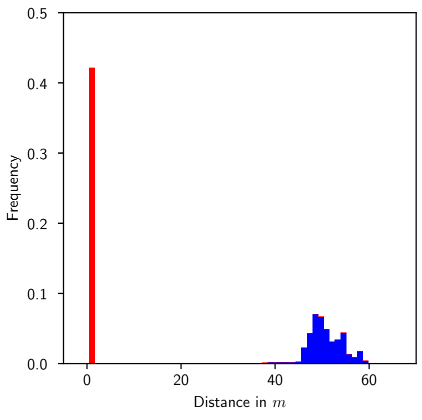
Figure A.8.: Weighted heat map of the absolute distance uncertainty of car 0 produced by the *UTM Extended ACC*. White areas indicate zero uncertainty, dark red areas indicate maximum uncertainty of 100 m.



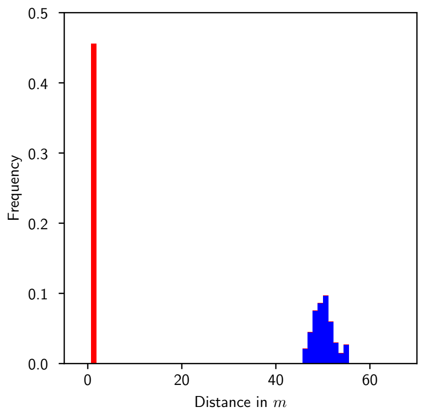
(a) Experiment **Basic**.



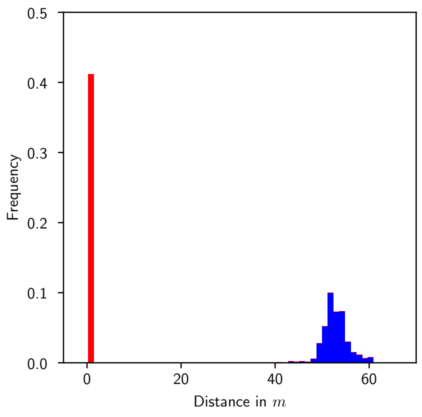
(b) Experiment **Ext**.



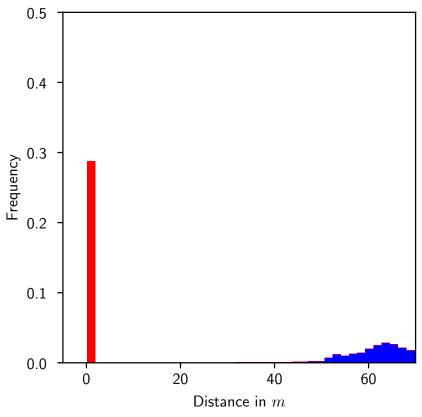
(c) Experiment **Pos1**.



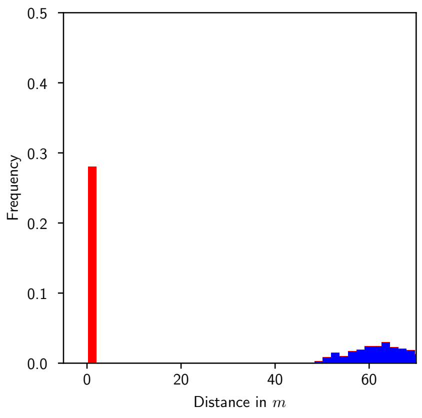
(d) Experiment **Pos2**.



(e) Experiment **Lossy**.

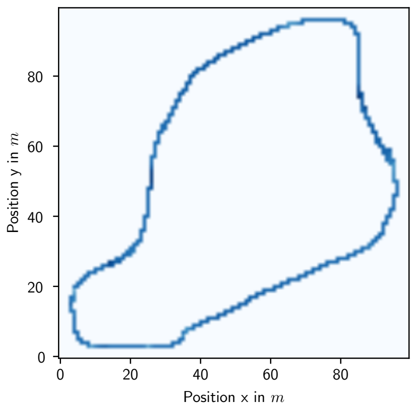


(f) Experiment **Late**.

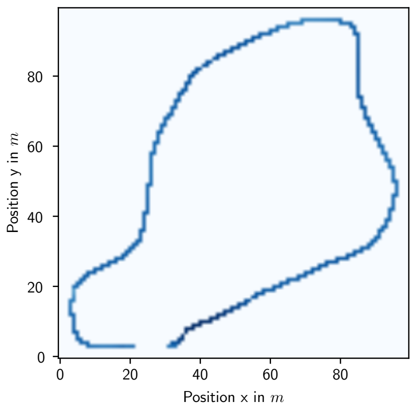


(g) Experiment **Bad**.

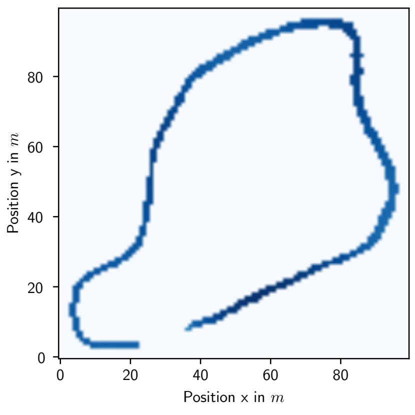
Figure A.9.: Comparison of the different road evaluation scenarios regarding **Distance** and **Distance** uncertainty of the first car as histograms produced by the *Road extended ACC*. Blue Bars indicate the **Distance** value and red bars indicate **Distance** uncertainty.



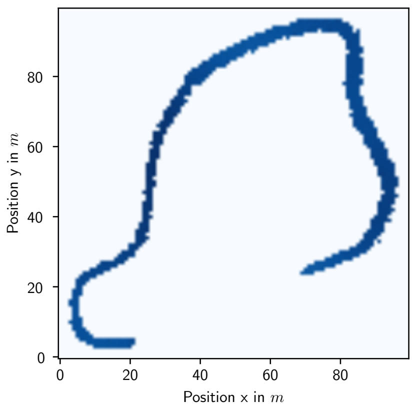
(a) Experiment **Basic**.



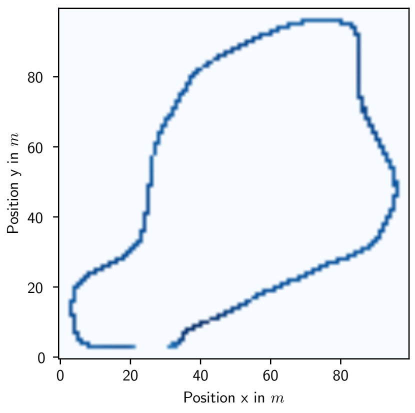
(b) Experiment **ExtR**.



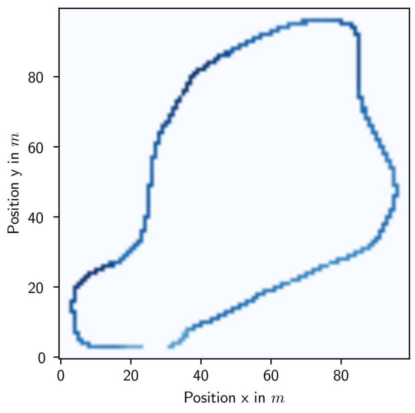
(c) Experiment **Pos1R**.



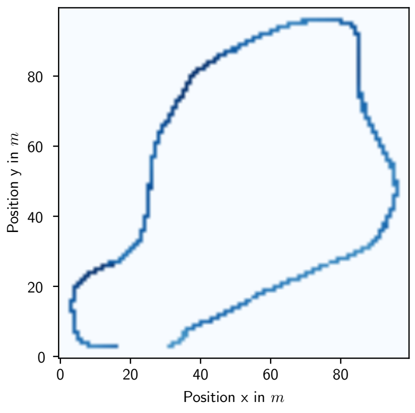
(d) Experiment **Pos2R**.



(e) Experiment **LossyR**.

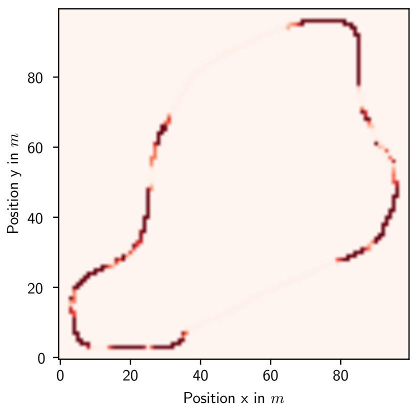


(f) Experiment **LateR**.

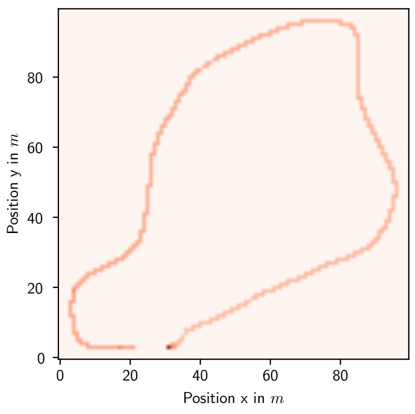


(g) Experiment **BadR**.

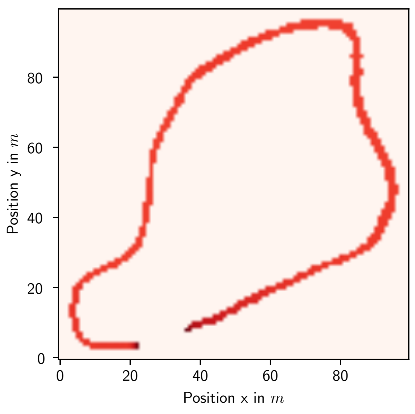
Figure A.10.: Weighted heat map of the absolute distance of car 0 produced by the *Road extended ACC*. White areas indicate zero distance, dark blue areas indicate maximum distance of 100 m.



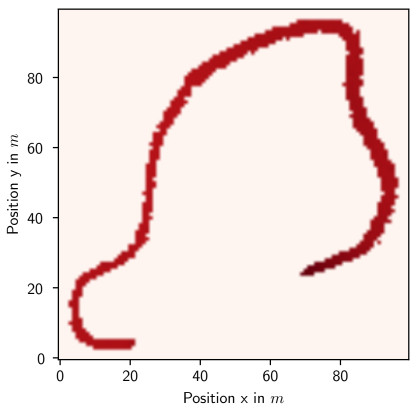
(a) Experiment **Basic**.



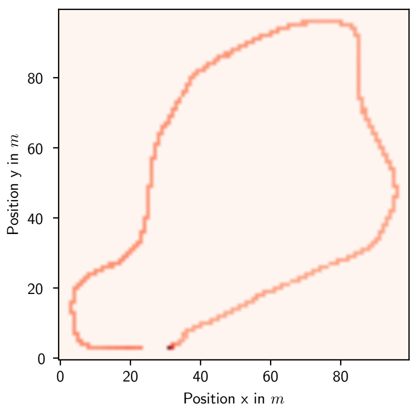
(b) Experiment **ExtR**.



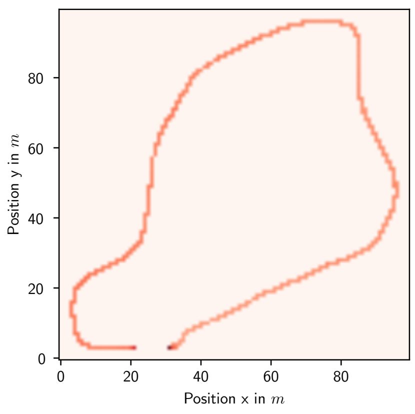
(c) Experiment **Pos1R**.



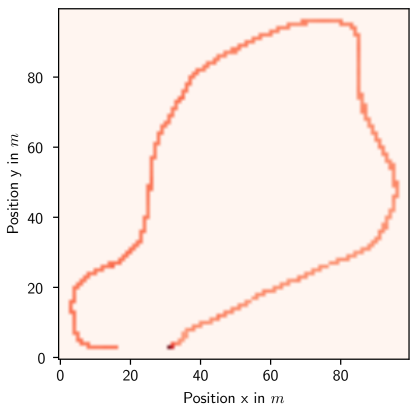
(d) Experiment **Pos2R**.



(e) Experiment **LossyR**.



(f) Experiment **LateR**.



(g) Experiment **BadR**.

Figure A.11.: Weighted heat map of the absolute distance uncertainty of car 0 produced by the *Road extended ACC*. White areas indicate zero uncertainty, dark red areas indicate maximum uncertainty of 100 m.

E h r e n e r k l ä r u n g

Ich versichere hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; verwendete fremde und eigene Quellen sind als solche kenntlich gemacht. Insbesondere habe ich nicht die Hilfe eines kommerziellen Promotionsberaters in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Ich habe insbesondere nicht wissentlich:

- Ergebnisse erfunden oder widersprüchliche Ergebnisse verschwiegen,
- statistische Verfahren absichtlich missbraucht, um Daten in ungerechtfertigter Weise zu interpretieren,
- fremde Ergebnisse oder Veröffentlichungen plagiiert,
- fremde Forschungsergebnisse verzerrt wiedergegeben.

Mir ist bekannt, dass Verstöße gegen das Urheberrecht Unterlassungs- und Schadensersatzansprüche des Urhebers sowie eine strafrechtliche Ahndung durch die Strafverfolgungsbehörden begründen kann. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form als Dissertation eingereicht und ist als Ganzes auch noch nicht veröffentlicht.

Magdeburg, den 08.11.2017

Dipl.-Inform. Christoph Steup