Flexible Processing of Streamed Context Data in a Distributed Environment

Von der Fakultät Informatik, Elektrotechnik und
Informationstechnik der Universität Stuttgart
zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Nazario Cipriani

aus San Giovanni / Italien

Hauptberichter:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Mitberichter:	Prof. Dr. rer. nat. Daniela Nicklas
Tag der mündlichen Prüfung:	06.06.2014

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart
2014

"I have always wished for my computer to be as easy to use as my telephone;
my wish has come true because I can no longer figure out how to use my telephone. "

— Bjarne Stroustrup.

Acknowledgements

When I first started my studies in Computer Science at the University of Stuttgart, I did not imagine to become a research staff member at the department of 'Applications of Parallel and Distributed Systems' supervised by Prof. Dr.-Ing. habil. Bernhard Mitschang. This thesis would have never become reality without the help of many people.

First of all, my deepest gratitude belongs to my doctoral advisor, Prof. Dr.-Ing. habil. Bernhard Mitschang. Thank you for giving me the opportunity to work in your research group on this challenging topic. Thank you for your guidance, your support and the many interesting discussions over all these years. Your comments gave me always valuable directions for my ongoing work.

Furthermore, my thanks go to the co-reviewer of my thesis, Prof. Daniela Nicklas, for spending time reading this document and giving valuable suggestions and comments.

I want to also thank all of my current and former colleagues at the department 'Applications of Parallel and Distributed Systems' for the interesting discussions and funny moments. I want to thank all of you! My special thanks go to: Andreas Brodt who shared the office with me over all the years; Tobias Kraft and Oliver Schiller for the coffee breaks, lunch breaks and interesting discussions; Christoph Stach for sharing his Nexus One with one of my students (and his tea with me); Matthias Großmann and Daniela Nicklas always having an open ear for me; Mihály Jakob and Marko Vrhovnik for the squash sessions; Peter Reimann, Fabian Kaiser, Clemens Dorda and Alexander Moosbrugger for the sporty table kicker sessions; Carlos Lübbe for supporting this work during his time as a student at our department.

Also, I want to particularly thank Holger Schwarz, Christoph Stach and Barbara Gommel for spending their precious free time reading this document and giving me valuable suggestions. My thanks also go to Friedemann Gschwind for suggestions concerning the section 'Deutsche Zusammenfassung'.

Further thanks go to all members and students at the faculty of Computer Science at the University of Stuttgart who supported me or contributed to this work. In, particular my thanks go to (in alphabetical order): Oliver Dörler, Raimund Huber, Benjamin Riehle, Daniel Garcia Sardina and Antonio Fernández Zaragoza.

I want to thank the German Research Foundation (DFG) and the State of Baden-Württemberg for funding my research. The results of this thesis were part of the SFB project with grant 627 entitled 'Nexus – Spatial World Models for Context-aware Applications'.

Thanks to all my friends and my family that were on my side over the years. Thank you Lukas Schikora, Peter Kaltstein and Steffen Maier for your open ears and patience. I would like to thank Lena Gschwind for her supporting words in difficult times. Heidi Raff and especially Euridike Adan-Döcker: Thank you for your advises during that time. Each one of you helped me a lot, even if you are not aware of that.

Last but not least, I want to also express my sincere thanks and gratitude to my parents Angela Cipriani and Mario Cipriani for their continuous support, encouragement and patience during my studies and during the work on this thesis. Thank you from the very bottom of my heart.

This work is dedicated to all of you, regardless of whether being small and big supporters.

Ostfildern, Dezember 12, 2013

Nazario Cipriani

Contents

List of Abbreviations

AC
Access Control

ACS
Access Control Service

ASR
Area Service Register

AWM
Augmented World Model

AWML
Augmented World Model Language

AWQL
Augmented World Query Language

CAP
Certificate Administration Point

CDC
Connected Device Configuration

CEO
Chief Executive Officer

CEP
Complex Event Processing

CGS
Core Graph Service

CLDC
Connected Limited Device Configuration

CML
Context Modeling Language

CPU
Central Processing Unit

CQL
Continuous Query Language

CS
Context Server

CSP
Constraint Satisfaction Problem

DB
Database

DBMS
Database Management System

DSL
Domain Specific Language

DSL
domain specific language

DSMS
Data Stream Management System

DSPS
Data Stream Processing System

EAI
Enterprise Application Integration

EAS
Extended Attribute Schema

EB
exabyte

ECS
Extended Class Schema

EMC
Execution Manager Client

EMS
Execution Manager Server

ER-model
Entity Relationship Model

ESB
Enterprise Service Bus

FPGA
Field Programmable Gate Array

GB
gigabyte

GC
Granularity Control

GHz
gigahertz

GPS
Global Positioning System

GPU
Graphics Processing Unit

GUI
graphical user interface

IAP
Identity Administration Point

ID
identifier

IFP
Information Flow Processing

J2ME
Java 2 Micro Edition

J2SE
Java 2 Standard Edition

JVM
Java Virtual Machine

KML
Keyhole Markup Language

kNN
k-nearest neighbors

LBS
location based service

LOD
Level of Detail

M-TOP
multi-target operator placement

MAC
Mandatory Access Control

MB
megabyte

MODB
Moving Objects Data Base

MRep
multiple represented object

MS
Monitoring Service

MSB
Manufacturing Service Bus

N statistics
node statistics

NEGM
Nexus Execution Graph Model

NN statistics
node-node statistics

NO statistics
node-operator statistics

NPGM
Nexus Plan Graph Model

NSL
Nexus Session Locator

OEE
Operator Execution Environment

OES
Operator Execution Service

OpS
Operator Scheduler

OQL
Object Query Language

ORC
Operator Repository Client

ORM
Object-Role Modeling

ORS
Operator Repository Service

OS
operating system

P2P
Peer 2 Peer

PAP
Policy Administration Point

PC
Process Control

PDP
Policy Decision Point

PKI
Public Key Infrastructure

QoS
Quality of Service

RAM
Random Access Memory

RAP
Role Administration Point

RBAC
Role Based Access Control

SAS
Standard Attribute Schema

SCC
Statistics Collector Client

SCS
Standard Class Schema

SFB 627
Sonderforschungsbereich 627

SIMD
Single Instruction Multiple Data

SL
: Service Loader

SLA
: Service Level Agreement

SM
: Service Manager

SMA
: Service Management Area

SMI
: Service Manager Interface

SOA
: Service Oriented Architecture

SOAP
: Simple Object Access Protocol

SP graph
: stream processing graph

SpaSe
: Spatial Model Server

SPGF
: SP-Graph Fragmenter

SPGI
: SP-Graph Interface

SPGO
: SP-Graph Optimizer

SPGP
: SP-Graph Planner

SQL
: Structured Query Language

SRS
: Service Repository Service

StaaS
: Streaming as a Service

STP
: Steiner-Tree Problem

STS
: Standard Type Schema

TAP
: Task Assignment Problem

TSP
: Traveling Salesman Problem

TTL
: time to live

UDF
: user-defined function

UML
: Unified Modeling Language

URI
: Uniform Resource Identifier

WS
: Web Service

WWS
: World Wide Space

WWW
: World Wide Web

XML
: Extensible Markup Language

Abstract

Nowadays, stream-based data processing occurs in many context-aware application scenarios, such as in context-aware facility management applications or in location-aware visualization applications. In order to process stream-based data in an application-independent manner, Data Stream Processing Systems (DSPSs) emerged. They typically translate a declarative query to an operator graph, place the operators on stream processing nodes and execute the operators to process the streamed data.

Context-aware stream processing applications often have different requirements although relying on the same processing principle, i. e. data stream processing. These requirements exist because context-aware stream processing applications differ in functional and operational behavior as well as their processing requirements. These facts are challenging on their own. As a key enabler for the efficient processing of streamed data the DSPS must be able to integrate this specific functionality seamlessly. Since processing of data streams usually is subject to temporal aspects, i. e. they are time critical, custom functionality should be integrated seamlessly in the processing task of a DSPS to prevent the formation of isolated solutions and to support exploitation of synergies.

Depending on the domain of interest, data processing often depends on highly domain-specific functionalities, e. g. for the application of a location-aware visualization pipeline displaying a three-dimensional map of its surroundings. The map displays the user's friends pinned to their current locations. The application runs on a mobile device and consists of many interconnected operations that form a network of operators called stream processing graph (SP graph). First, the friends' locations must be collected and connected to their public profile. To get a nicely displayed map, beside the streamed data of many mobile objects, i. e. the friends, the application needs to integrate also—rather static—map data. Finally, the scene must be rendered and displayed on the mobile device. However, to enable the application to run smoothly for some parts of data processing the presence of a Graphics Processing Unit (GPU) is mandatory.

To solve that challenge, we have developed concepts for a flexible DSPS that allows the integration of specific functionality to enable a seamless integration of applications into the DSPS. Therefore, an architecture is proposed. A DSPS based on this architecture can be extended by integrating additional operators responsible for data processing and services realizing additional interaction patterns with context-aware applications. However, this specific functionality is often subject to deployment and run time constraints. Therefore, an SP graph model has been developed which reflects these constraints by allowing to annotate the graph by constraints, e. g. to constrain the execution of operators to only certain processing nodes or specify that the operator necessitates a GPU.

The data involved in the processing steps is often subject to restrictions w.r.t the way it is accessed and processed. Users participating in the process might not want to expose their current location to potentially unknown parties, restricting e. g. data access to known ones only. Therefore, in addition to the flexible integration of specialized operators security aspects

must also be considered, limiting the access of data as well as the granularity of which data is made available. We have developed a security framework that defines three different types of security policies: Access Control (AC) policies controlling data access, Process Control (PC) policies influencing how data is processed, and Granularity Control (GC) policies defining the Level of Detail (LOD) at which the data is made available. The security policies are interpreted as constraints which are supported by augmenting the SP graph by the relevant security policies.

The operator placement in a DSPS is very important, as it deeply influences SP graph execution. Every stream-based application requires a different placement of SP graphs according to its specific objectives, e. g. *bandwidth should not fall below 500 MBit/s and is more important than latency*. This fact constrains operator placement. As objectives might conflict among each other, operator placement is subject to trade-offs. Knowing the bandwidth requirements of a certain application, an application developer can clearly identify the specific Quality of Service (QoS) requirements for the correct distribution of the SP graph. These requirements are a good indicator for the DSPS to decide how to distribute the SP graph to meet the application requirements. Two applications within the same DSPS might have different requirements. E. g. if interactivity is an issue, a stream-based game application might in a first place need a minimization of latency to get a fast and reactive application. We have developed a multi-target operator placement (M-TOP) algorithm which allows the DSPS to find a suitable deployment, i. e. a distribution of the operators in an SP graph which satisfies a set of predefined QoS requirements. Thereby, the M-TOP approach considers operator-specific deployment constraints as well as QoS targets.

Finally, to increase the usability of the system and to enable its full capabilities developers should be given appropriate tool support to facilitate the development process and reduce the danger of erroneous development settings. Therefore, we have developed tool support for the design and development of stream-based context-aware applications. The tool supports the developers during application development process by providing features such as assistance for operator development and a graphical user interface for the composition of SP graphs.

In conclusion, this thesis presents concepts and solutions to build an extensible DSPS infrastructure that enables developers of context-aware applications and its users to take advantage of data stream processing capabilities and at the same time to be able to tailor the DSPS functionalities to meet their requirements.

Deutsche Kurzfassung

1 Einleitung und Motivation

Das Datenvolumen, mit dem wir uns im Alltag konfrontiert sehen, nimmt stetig zu. Eric Schmidt, ehemaliger Geschäftsführer (engl. Chief Executive Officer (CEO)) von Google, gab eine Schätzung des Datenbestands des Internets ab und bezifferte diesen auf eine Zahl von 5 exabyte (EB) [27]. Laut Eric Schmidt sehen wir uns bereits heute einer enormen Datenflut ausgesetzt, wobei alle zwei Tage so viele Daten anfallen, wie die gesamte Menschheit im gesamten Jahr 2009 generierte. Diese Daten fallen hierbei in unterschiedlichen Datenquellen an: Im Internet mit all seinen (typischerweise) unstrukturierten Daten, in Datenbanken, in denen Daten in strukturierter Form vorliegen sowie in Sensoren, welche sich in unserer Umgebung befinden und die Umwelt erfassen und einen kontinuierlichen Datenstrom generieren. Eben diese Tatsache macht es schwierig der Lage Herr zu werden. Denn zum Problem der schieren Datenmenge kommen noch Aspekte der Heterogenität und der Volatilität der Daten hinzu [56]. Es gibt bereits eine Menge interessanter Vorschläge zur Verarbeitung strukturierter und unstrukturierter Daten, welche sich jedoch meist auf Daten statischer Natur beschränken. Statisch bedeutet in diesem Zusammenhang, dass es sich um Daten handelt, die sich mit einer niedrigen Frequenz verändern und somit einer relativ geringen Veränderungsrate unterliegen. Dagegen stellen sich bezüglich der Verarbeitung von Datenströmen, welche naturgemäß einer sehr hohen Änderungsrate unterliegen und die zudem als zeitkritisch anzusehen sind, viele offene Fragen. Der Forschungsbedarf rührt vor allem daher, dass sich in diesem Fall eine vorherige Speicherung aller relevanter Daten in beispielsweise einer Datenbank und deren anschließende Verarbeitung durch die Annahme der potentiellen Unendlichkeit der Datenströme verbietet. Zudem unterstützen Datenbankmanagementsysteme üblicherweise nicht die effiziente fortlaufende Verarbeitung großer Datenmengen durch kontinuierliche Anfragen (sog. *continuous queries* [133]) da diese hierfür nicht ausgelegt sind.

Ein Datenstrom charakterisiert sich unter anderem durch seinen potentiell unendlichen Fluss von Datenelementen aus einer Quelle. Die Verarbeitung solcher Daten stellt eine große Herausforderung dar und wird typischerweise in zwei Schritten vollzogen: Die Datenelemente der Datenquellen werden abschnittsweise gesammelt und anschließend durch eine zuvor definierte Reihenfolge an Operationen verarbeitet. Diese lassen sich anschaulich durch ein Netzwerk von miteinander verbundenen Operatoren beschreiben, welche eine Verarbeitungs-Pipeline definieren. In der jüngsten Vergangenheit lag Datenstromverarbeitung im Fokus vieler wissenschaftlicher Arbeiten der Datenbank-Fachleute. Die erarbeiteten Vorschläge reichen hierbei von Aufbauvorschlägen für Datenstromverarbeitungssysteme wie beispielsweise [2, 78, 129], über spezifische Datenstromverarbeitungstechniken wie die Verwendung von Interpunktionen [141] zur effizienten Verarbeitung von Daten oder die Migration von Operatoren bei Überlastsituationen auf einem bestimmten Rechenknoten [151, 155], bis hin zur Frage der effizienten verteilten Ausführung von Verarbeitungsgraphen [112, 126, 153].

In dieser Arbeit soll es um die datenbankbezogene Perspektive auf das Thema der Datenstromverarbeitung gehen, welche erstmals von Babcock et al. [16] erwähnt wurde. In Babcock et al. [16] geht es um die Frage, wie ein Management-System zur Verarbeitung von Datenströmen aufgebaut ist. Der Fokus besteht also in der Übertragung bekannter Konzepte aus dem Bereich der auf dem sog. *Pull-Paradigma* fußenden Datenbankmanagementsysteme in den Kontext eines Management-Systems zur Verarbeitung von Datenströmen, welche auf das andersartige sog. *Push-Paradigma* fußen. Die Idee der Datenstromverarbeitungssysteme ist die Adressierung der effizienten Verarbeitung von großen und nicht zuvor speicherbaren Datenbeständen, welche aber gleichzeitig einer hohen Fluktuationsrate unterliegen können und somit als zeitkritisch zu betrachten sind. Üblicherweise operieren die Systeme hierbei auf strukturierten Daten. Jedoch erlauben es gewisse Systeme, auch heterogene Datenbestände zu verarbeiten, wie SystemS [9] oder das in dieser Arbeit vorgestellte NexusDS. Datenstromverarbeitungssysteme haben es dennoch nicht geschafft eine große Nutzergemeinde anzusprechen, wie es seinerzeit die Datenbanktechnologie tat. Das Problem hierbei ist, dass abhängig von der konkreten Domäne, wie beispielsweise die kontextbezogene Visualisierung von Umgebungsmodelldaten, dedizierte Berechnungsvorschriften und Methoden gefordert sind. Diese dedizierte Funktionalität spiegelt sich unter Anderem in spezifischen Operatoren wider, welche unter Umständen eine spezifische Umgebung benötigen, um effizient laufen zu können, wie beispielsweise eine GPU. Zudem sollte man unter Anderem aufgrund der hohen Datenmengen eine enge Integration der spezifischen Funktionalitäten anstreben, da dies sonst zu Engpässen oder ineffizienter Verarbeitung führen kann. Dies resultiert in einer sehr hohen Abhängigkeit und somit in einem engen Grad der Kopplung zwischen den Anwendungsbedürfnissen auf der einen und den Systemfähigkeiten auf der anderen Seite [45]. Diese Anwendungsbedürfnisse müssen von Datenstromverarbeitungssystemen berücksichtigt werden. In diesem Zusammenhang stellen insbesondere *kontextbezogene Anwendungen* eine große Herausforderung dar, wie zum Beispiel eine kontextbezogene Visualisierungsanwendung. Diese basiert auf einem Modell der realen Welt, die aus Daten zusammengesetzt wird, welche aus unterschiedlichen Datenquellen stammen und eine dreidimensionale Karte der näheren Umgebung zeichnet. Hierbei fließen sowohl dynamische Informationen vieler hundert Sensoren ein, die beispielsweise die aktuelle Position mehrerer mobiler Objekte im Sichtfenster ermitteln, als auch statische Informationen, welche beispielsweise das Modell der Stadt liefern in der man sich befindet.

Man könnte nun argumentieren, dass die Funktionalität, welche von kontextbezogenen Anwendungen benötigt wird, ohnehin entwickelt werden muss, um die entsprechende Anwendung zu realisieren. Das stimmt zwar, dennoch gibt es große Vorteile, diese in einem Datenstromverarbeitungssystem zu integrieren. Schaut man auf die Forschung der letzten zehn Jahre zurück, stellt man fest, dass sich die Forschungsarbeit hauptsächlich auf die Entwicklung hocheffizienter Lösungen konzentrierte, welche in einer von Anwendungen unabhängigen Weise die effiziente Verarbeitung von Datenströmen erlauben soll. Dies gilt allerdings nicht für die Domäne der kontextbezogenen Datenstromanwendungen. Es war nicht vorgesehen, auf die Notwendigkeiten solcher Anwendungsklassen einzugehen. Allerdings fußen diese Anwendungen auf hochspezialisierten Verarbeitungs- und Interaktionsmustern wie beispiels-

weise das Zeichnen einer Szenerie der näheren Umgebung und zugleich deren Anpassung an Benutzerinteraktionen. Zudem unterstützen vorhandene Datenstromverarbeitungssysteme typischerweise nicht die Verarbeitungsschritte der Daten auf einer heterogenen Systemtopologie auszuführen und so beispielsweise spezialisierte Hardware auszunutzen. Ein Anwendungsentwickler müsste nun anfangen sich für die anfallenden spezialisierten Funktionen zu überlegen, wie er diese konkret umsetzt. Hierbei ist es fast unvermeidlich, dass Insellösungen in der Form entstehen, dass spezialisierte Funktionalität in einen (vom Datenstromverarbeitungssystem) unabhängigen Bereich installiert und ausgeführt werden. Das hat zur Folge, dass sich beispielsweise lange Transportwege womöglich hoher Datenvolumina ergeben und somit die Verwendung potentiell nützlicher Synergieeffekte ausbleibt oder erschwert wird. Um die Inselbildung und somit potentielle Redundanzen zu vermeiden, müssen solche Systeme eine enge Integration der Anwendungsfunktionalität mit der eigentlichen Datenverarbeitung anstreben. Hierbei ergibt sich ein hohes Synergie- und Kosteneinsparungspotential. Durch die Integration spezialisierter Funktionalität lassen sich beispielsweise lange und unnötige Transportwege für die zu verarbeitenden Daten vermeiden.

Diese Arbeit hat den Entwurf und die Implementierung eines erweiterbaren und flexiblen Datenstromverarbeitungssystems zum Ziel, welches auf die speziellen Bedürfnisse kontextbezogener Anwendungen zugeschnitten ist, dabei entsprechende Freiheitsgrade erlaubt und die oben beschriebenen Probleme vermeidet. Im Speziellen leistet diese Arbeit folgende Beiträge: *Motivation der Notwendigkeit* eines solchen Systems mit Hinblick auf Problemformulierung und Vergleich mit bestehenden Lösungsvorschlägen, Architektur eines skalierbaren und flexibel erweiterbaren *Datenstromverarbeitungssystems*, Definition eines Restriktionsraums sowie des zugehörigen *Restriktionsmodells* und deren konkrete Umsetzung im Datenstromverarbeitungssystem, ein *Sicherheitskonzept*, womit der Zugriff auf Daten sowie deren Verarbeitung bestimmt oder beeinflusst werden kann, eine *automatisierte Platzierung* von Verarbeitungsgraphen auf die zur Verfügung stehenden Verarbeitungsknoten, sowie der Architektur einer maßgeschneiderten *Werkzeugunterstützung* für das im Rahmen dieser Arbeit entwickelte Datenstromverarbeitungssystem zur Unterstützung der Anwendungsentwickler.

Der Rest der deutschen Zusammenfassung gliedert sich wie folgt: In Kapitel 2 werden die grundsätzlichen Anforderungen anschaulich am Beispiel verschiedener Anwendungen erläutert und zusammengefasst. Im darauffolgenden Kapitel 3 wird die Architektur von NexusDS sowie die Mechanismen vorgestellt, welche eine flexible Integration und Erweiterung des Systems erlauben. NexusDS ist das Datenstromverarbeitungssystem, welches im Rahmen dieser Arbeit entstanden ist und die hier vorgestellten Konzepte umsetzt. In Kapitel 4 wird das Dienst- und Operator-Framework erläutert und die Ausführung der Verarbeitungsgraphen vorgestellt. In Kapitel 5 wird das Sicherheitskonzept von NexusDS vorgestellt, das es ermöglicht, den Zugriff auf Daten sowie die Verarbeitungscharakteristiken zu beeinflussen. In Kapitel 6 wird die von NexusDS vorgenommene automatisierte Verteilung der Verarbeitungsgraphen vorgestellt. Hierbei werden sowohl heterogene Systemtopologien berücksichtigt als auch die Restriktionen, welche beispielsweise von Entwicklern vorgegeben oder Anwendern verfeinert werden können. Kapitel 7 stellt die Werkzeugunterstützung für NexusDS sowie ausgewählte Anwen-

dungen als Machbarkeitsnachweis vor. Abgeschlossen wird die deutsche Zusammenfassung in Kapitel 8 durch eine kurze Zusammenfassung und einem Ausblick auf mögliche zukünftige Arbeiten.

2 Grundlagen und Anforderungen

Kapitel 2 behandelt die Grundlagen für diese Arbeit, erörtert verschiedene Anwendungsszenarien kontextbezogener Anwendungen und definiert Anforderungen an ein flexibles Datenstromverarbeitungssystem.

Kontextbezogene Anwendungen bringen Anforderungen mit sich, welche von bisherigen Datenstromverarbeitungssystemen nicht bedient werden können. Hier gibt es abhängig von der jeweiligen Domäne wie beispielsweise der Domäne der Visualisierung eine starke Abhängigkeit zu hoch domänenspezifischer Funktionalität. Viele dieser Anwendungen fußen dabei auf der gleichen Verarbeitungstechnik: Datenstromverarbeitung. Aus diesem Grund ist es sinnvoll, vorhandene Funktionalität wo möglich wiederzuverwenden und so ein hohes Einsparpotential zu erreichen. Dazu müssen allerdings eine Reihe von Anforderungen erfüllt werden. Wie in [43] beschrieben, besteht eine Notwendigkeit für Datenstromverarbeitungssysteme darin, sich an die Anforderungen kontextbezogener Anwendungen anzupassen. Das entsprechende Datenstromverarbeitungssystem muss Möglichkeiten anbieten, die jeweiligen Besonderheiten geeignet ausdrücken zu können und hierfür entsprechende Mechanismen bereitstellen, welche dann die Anforderungen entsprechend umsetzen. Ein Beispiel für eine kontextbezogene Anwendung, welche auch als Prototyp in Rahmen dieser Arbeit umgesetzt wurde, ist eine Visualisierungsanwendung, welche Luftströmungen innerhalb eines Gebäudes anzeigt und die aktuelle Position mobiler Objekte wie auch statischer Objekte im Gebäude bei der Berechnung berücksichtigt.

Das Datenstromverarbeitungssystem muss also eine Möglichkeit zur Erweiterung bieten. Zusätzlich muss noch gewährleistet sein, dass die entsprechende Software und Hardware-Konfiguration auf dem Zielrechner vorhanden ist, auf dem die Anwendung (oder Teile davon, wenn man von einer verteilten Ausführung ausgeht) laufen soll. Im oben beschriebenen Szenario benötigt die Anwendung tatsächlich eine GPU für den sogenannten Render-Prozess, welcher für die Berechnung des darzustellenden Bildes zuständig ist. Die korrekte Ausführung ist also auf eben solche Komponenten beschränkt. Es muss also eine Verarbeitungskomponente vorhanden sein, welche die notwendige Funktionalität (GPU) bietet. Da es hierbei eine vielzahl verschiedener Komponenten geben kann, führt das zu einer heterogenen Systemtopologie. Darüber hinaus gibt es auch weitere Restriktionen. Stellt man sich eine solche Anwendung wie oben beschrieben in einem Fabrikszenario vor, wird schnell deutlich, dass bestimmte Daten nicht über bestimmte Bereiche hinausgetragen werden sollen. Darüber hinaus ist es wichtig zu wissen, wer auf welche Daten zugreift. Man benötigt in der Regel einen entsprechenden Zugriffsmechanismus, um das Lesen oder Schreiben der Daten zu kontrollieren, da die Daten zusätzlich bestimmten Zugriffs- und Verarbeitungsrestriktionen unterliegen. Aus diesem Grund

ist es notwendig, den Transport- und Verarbeitungsradius für Daten einschränken zu können, aber auch den kontrollierten Zugriff darauf zu gewährleisten. Um den unnötigen Transport von großen Datenmengen zu vermeiden ist es zudem sinnvoll, die Verarbeitung (falls möglich) nahe an die Stellen zu verlagern, an der die Daten anfallen. Im Falle einer Trajektorienkompressionsanwendung fallen die Daten beispielsweise als Datenstrom von Positionen eines mobilen Geräts an. Hierfür dient das mobile Gerät einmal als Datenquelle, welche die eigentlichen Positionsinformationen liefert. Die Übertragung aller Positionsdaten jedoch ist nicht immer wünschenswert. Das hat zum einen einen datenschutzrechtlichen Aspekt und zum anderen den Aspekt unnötige Übertragungen zu vermeiden, um Bandbreite und Energie einzusparen. Aus diesen Gründen macht es beispielsweise Sinn, das Gerät in die Verarbeitung einzubinden und einen Teil der Verarbeitung auf das mobile Gerät zu verlagern.

Zusammenfassend lassen sich die folgenden Anforderungen formulieren: *Maßgeschneiderte Datenverarbeitung* durch Integration neuer Operatoren und Dienste in das Datenstromverarbeitungssystem, *Unterstützung sowohl strukturierter als auch unstrukturierter Datenverarbeitung sowie die integrierte Verarbeitung statischer als auch dynamischer Daten*, *Berücksichtigung von Verteilungs- und Ausführungsrestriktionen* beispielsweise zur Definition sicherheitsrelevanter Restriktionen, *Unterstützung einer heterogenen Systemtopologie*, da die an Berechnungen teilnehmenden Rechenknoten unterschiedliche Konfigurationen haben können und die Verwendung *mobiler Geräte sowohl als Datenquelle als auch als Ausführungseinheiten*. Ferner müssen Aspekte bezüglich des *kontrollierten Zugriffs* und der *kontrollierten Verarbeitung* von Daten betrachtet werden.

Um diese Datenstromszenarien tatsächlich in einem Datenstromverarbeitungssystem umzusetzen, müssen entsprechende Mechanismen und Konzepte geschaffen werden. Im Folgenden wird die Systemarchitektur von NexusDS vorgestellt. NexusDS ist ein Datenstromverarbeitungssystem, welches auf die Bedürfnisse der oben skizzierten Anwendungen zugeschnitten ist und die aufgezeigten Anforderungen erfüllt.

3 Systemarchitektur

Kapitel 3 beschreibt die Architektur und Funktionsweise des im Rahmen dieser Arbeit entstandenen Datenstromverarbeitungssystems, welches die im vorangegangenen Kapitel genannten Anforderungen adressiert. Zudem wird ein Konzept zur Integration statischer Daten in die Datenstromverarbeitung vorgestellt.

Die grundlegenden Problemstellungen hinter der Datenstromverarbeitung wurden erstmals 2002 benannt und sind in [16] und [60] beschrieben. Darauf folgend gab es eine Vielzahl an Arbeiten, und viele Konzepte und Ideen aus der Datenbankwelt fanden Einzug in den Bereich der Datenstromsysteme, wenn auch, bedingt durch die Besonderheiten bei der Datenstromverarbeitung, oft in abgewandelter Form. Die grundsätzliche Idee hinter Datenstromverarbeitungssystemen ist es, statt langlebiger Daten und kurzlebiger Anfragen, wie es bei klassischen Datenbanken der Fall ist, kurzlebige Daten und langlebige Anfragen zu betrachten. Hierbei

spricht man auch von einem Paradigmenwechsel, weg von einem Pull-basierten Modell (im Falle von Datenbanken) hin zu einem Push-basierten Modell (im Falle von Datenstromsystemen).

Datenstromverarbeitungssysteme lassen sich grob in zentralisierte und verteilte Systeme unterteilen. Die ersten Datenstromverarbeitungssysteme waren zentralisiert wie beispielsweise Aurora [3], STREAM [14] oder PLACE [96]. Deren großer Vorteil ist, dass die Verarbeitung effizient und schnell von statten geht, da keine Rechnergrenzen überschritten werden müssen. Jedoch barg dieser Ansatz auch Probleme, nämlich dass die Systeme nicht mit der Anzahl der gestellten Anfragen bzw. der zu bewältigenden Last skalieren konnten. Die zweite Generation waren verteilte Systeme, welche die Möglichkeit boten, die Verarbeitung verteilt zu verrichten. Repräsentanten für diese Systemklasse sind beispielsweise Borealis [2], StreamGlobe [80], SystemS [57] oder PIPES [79]. Dabei hatte jedes System einen anderen Fokus: SystemS [57] hatte als Hauptanwendung Data-Mining-Anwendungen, StreamGlobe [80] fokussierte sich auf Katastrophenszenarien, PLACE* [150] hatte als Hauptanwendung die Verwaltung mobiler Objekte, PIPES [79] war ein generisches Framework zur Erstellung eines Datenstromverarbeitungssystems und Borealis [2] hatte als Hauptanwendung Applikationen, welche auf die Verknüpfung von Datenströmen und DB-Daten beruhten.

NexusDS ist ein verteiltes Datenstromverarbeitungssystem, das speziell auf die im vorangegangenen Abschnitt vorgestellten Bedürfnisse kontextbezogener Anwendungen zugeschnitten ist. Im Folgenden wird die Architektur von NexusDS sowie dessen grundlegenden Konzepte vorgestellt.

NexusDS baut auf dem Nexus-System [103] auf, einer offenen Plattform für kontextbezogene Anwendungen. Nexus beruht auf der Annahme, dass Daten in unterschiedlichen Datensilos und von unterschiedlichen Datenanbietern zur Verfügung gestellt werden. Diese Daten werden durch eine Föderation zusammengeführt und über einen Anfrage-Antwort-Mechanismus zugreifbar gemacht. Wie auch Nexus, baut NexusDS auf dem Augmented World Model (AWM) auf. Das AWM ist ein erweiterbares und auf objektorientierten Prinzipien beruhendes Datenmodell [105]. Das AWM definiert Klassen, wie beispielsweise Gebäude, Straßen oder sog. "*Points of Interests*" wie auch deren Attribute. Zu den Attributen können noch zusätzlich Meta-Attribute definiert werden, welche weitere Informationen zu den Attributwerten angeben. Ein AWM-Objekt kann mehrere Attributinstanzen desselben Attributs haben, wie zum Beispiel Positionsinformationen. Mit den Metadaten, welche sich zu den jeweilige Attributinstanzen definieren lassen, kann jeder Positionsinformation beispielsweise ein eindeutiger Zeitstempel zugeordnet werden. Das führt zu der nützlichen Eigenschaft, dass sich die einzelnen Attributinstanzen in eine eindeutige (zeitliche) Reihenfolge bringen lassen, sofern die Zeitstempel eindeutig sind.

Die Architektur von NexusDS besteht aus fünf Schichten, wie in Abbildung 1 dargestellt. NexusDS kombiniert eine flexible Dienstplattform und ein Operator-Framework, welche es erlauben Zusatzfunktionalitäten nach Bedarf zur Verfügung zu stellen. Das Hauptanliegen von NexusDS ist die einfache und nahtlose Integration von operationalen und funktionalen Erweiterungen in Form von Operatoren und Diensten, um so die Integration domänen- und anwen-

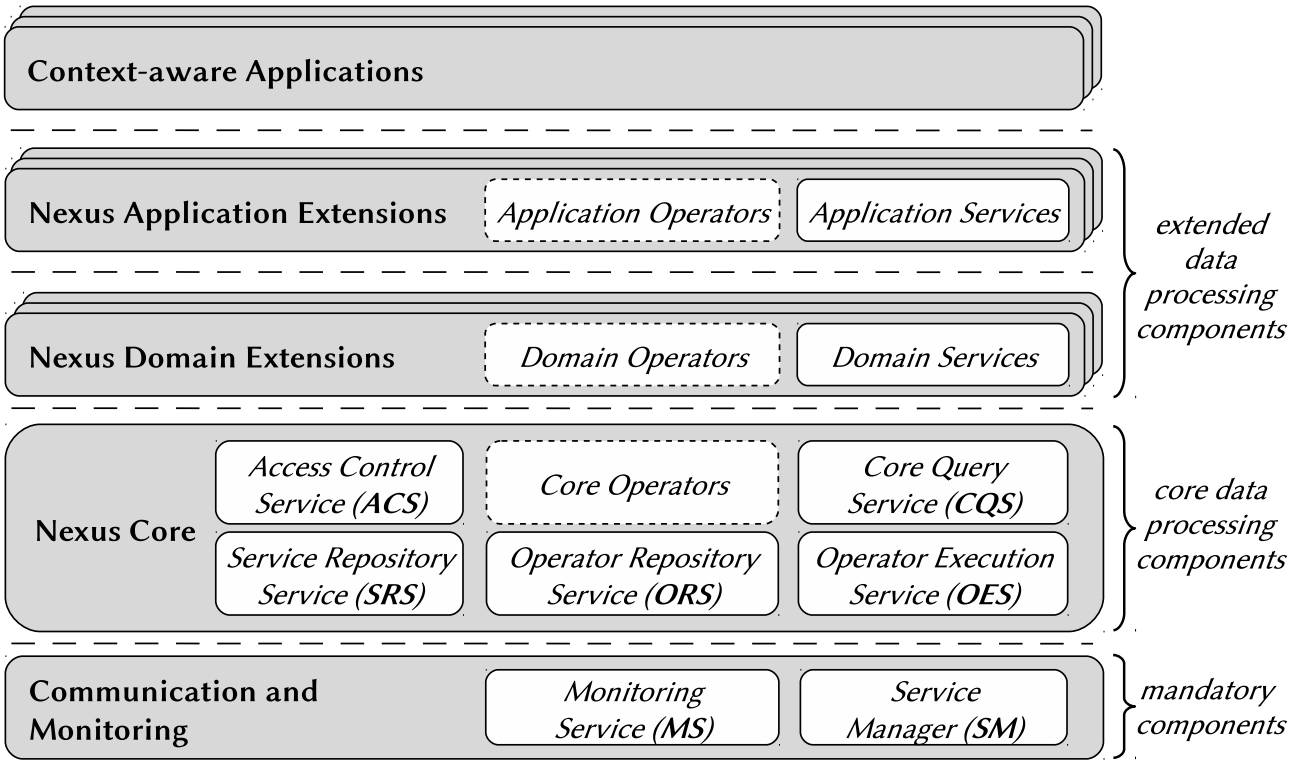


Abbildung 1: Schichtenarchitektur von NexusDS

dungsspezifischer Funktionalität zu gewährleisten. Ein zusätzlicher Operator wäre zum Beispiel der zuvor aufgeführte Render-Operator zum Zeichnen einer dreidimensionalen Karte der näheren Umgebung. Ein zusätzlicher Dienst wäre zum Beispiel ein Visualisierungsdienst, welcher eine spezialisierte Datenverarbeitungsdefinitionssprache zur Verfügung stellt, die auf die Domäne der Visualisierung zugeschnitten ist. Dadurch könnten beispielsweise maßgeschneiderte domänenspezifische Sprachen, sog. Domain Specific Languages (DSLs) in das System integriert werden. Den Operatoren (als gestrichelte Kästen in Abbildung 1 dargestellt) liegt ein Push-basiertes Paradigma zugrunde. Es dient dazu, die Datenverarbeitung vorzunehmen. Hierbei wurde ein besonderes Augenmerk darauf gelegt, dass die Integration neuer Operatoren möglichst einfach funktioniert und sich somit die Datenverarbeitung beliebig erweitern lässt. Dem gegenüberstehend sind Dienste (in durchgezogenen Boxen dargestellt) Pull-basiert, was bedeutet, dass zu jeder Anfrage eine Antwort erfolgt und dann die Verarbeitung beendet ist. Die Dienste sind nach dem Prinzip der Service Oriented Architecture (SOA) lose gekoppelt und können mit anderen Diensten interagieren. Dienste dienen dazu, ein spezifisches Dienstangebot in das System zu integrieren und so beispielsweise einen spezialisierten Visualisierungsdienst anbieten zu können. Dieser neue Dienst wird nahtlos in die vorhandene Systemarchitektur integriert und stellt zusätzliche Dienstfunktionalität zur Verfügung. Dieser Dienst könnte dann die gestellten Anfragen so verändern, dass es den jeweiligen Richtlinien (beispielsweise Ausführung in bestimmten Umgebungen) entspricht oder spezialisiertes Optimierungspotential ausgeschöpft werden kann (da der Dienst auf jeweilige Domäne angepasst ist und dies in seiner Ausführungssemantik berücksichtigt).

4 Verarbeitungsgraph

Kapitel 4 beschäftigt sich mit dem Konzept zur flexiblen Definition von Verarbeitungsgraphen, der Einbindung statischer Daten in die Datenstromverarbeitung sowie der Erweiterbarkeit des Systems.

Üblicherweise bieten Datenstromverarbeitungssysteme eine deklarative Anfrageschnittstelle, die bekannteste hierfür ist die Continuous Query Language (CQL) von Arasu et al. [15]. Die CQL-Syntax ist hierbei stark an die von Structured Query Language (SQL) angelehnt und wurde um für die Datenstromverarbeitung relevante Statements erweitert. Beispiele hierfür sind die Definition von Datenfenstern oder die Angabe der Ausführungsdauer. Allerdings lässt sich eine deklarative Anfragesprache nur schwer durch zusätzliche Operatoren erweitern. Andere Definitionssprachen sind wiederum programmatischer Natur und richten sich hauptsächlich an Programmierer. Ein Beispiel hierfür ist das von Gedik et al. [57] vorgeschlagene Modell namens *SPADE*. Hierbei erstellt der Entwickler ein Programm, das im Anschluss durch einen Compiler in einen entsprechenden Operatorgraphen übersetzt wird. Dieser Ansatz bietet zwar die Flexibilität zusätzliche Operatoren einzubringen, setzt aber die Komposition einer entsprechenden Orchestrierung der Datenverarbeitung und Kenntnis im Umgang mit Programmierung voraus.

Das Modell in NexusDS zur Orchestrierung der Datenverarbeitung besteht aus einem flexiblen Boxenprinzip, genannt Nexus Plan Graph Model (NPGM). Eine Box stellt hierbei eine Verarbeitungsoperation wie *Filterung* dar. Durch die Verknüpfung mehrerer Boxen ergibt sich ein Verarbeitungsgraph, der die Verarbeitungsreihenfolge der Daten beschreibt. Ähnlich wie von Cherniack et al. [38] vorgeschlagen, werden hierbei verschiedene Boxen durch Verbindungen zusammengebracht. Jede dieser Verbindungen repräsentiert eine Datenabhängigkeit zwischen zwei Boxen. Im Unterschied zu den genannten Ansätzen bietet das NPGM eine intuitive Möglichkeit, die entsprechende Datenverarbeitung grafisch zu definieren, ist beliebig durch zusätzliche Boxentypen erweiterbar und erlaubt zusätzlich Restriktionen mitzugeben, um so das tatsächliche Deployment beeinflussen zu können. Eine Box kann hierbei entweder ein Operator, eine Quelle oder eine Senke sein. Die Datenstromverarbeitung beginnt bei den Quellen und geht über die Operatoren in der vom Operatorgraphen definierten Reihenfolge. Die Verarbeitung endet bei den Senken. Es gibt hierbei zwei unterschiedliche Restriktionsklassen: Deployment-Restriktionen und Ausführungs-Restriktionen. Deployment-Restriktionen wirken sich zur Zeit des Deployments aus und bedingen eine entsprechende Wahl des Deployments, wie beispielsweise Selektion eines bestimmten Rechenknotens. Ausführungszeit-Restriktionen bedingen das Deployment, wirken sich aber letztendlich auf die tatsächliche Laufzeit aus, wie beispielsweise bei der Wahl einer bestimmten Auflösung eines Visualisierungsoperators.

In NPGM dargestellte Verarbeitungsgraphen sind jedoch noch nicht ausführbar, da die Definition auf einer logischen Ebene stattfindet. Das bedeutet beispielsweise, dass keine konkreten Implementierungen für die jeweiligen Boxen angegeben werden, sondern stattdessen eine Operatorklasse wie "Render". Um jedoch eine ausführbare Repräsentation des Verarbeitungsgraphen zu bekommen, muss der gesamte NPGM-Graph noch um die fehlenden Angaben er-

gänzt werden. Das Modell der ausführbaren Repräsentation eines NPGM-Graphen nennt sich Nexus Execution Graph Model (NEGM) und stellt eine ausführbare und stabile, sprich "deploybare", Definition des ursprünglichen NPGM-Graphen dar. Diese Abbildung läuft in mehreren Schritten ab: Zunächst wird überprüft, ob Zugriffsrichtlinien eingehalten werden. In einem zweiten Schritt wird versucht, Selektionen möglichst nah an die Datenquellen zu verschieben, um unnötigen Datentransfer zu verhindern. Danach wird durch einen Fragmentierer der NPGM-Graph in einen NEGM-Graphen transformiert, so dass er in einem letzten Schritt auf die entsprechenden Ausführungsumgebungen verteilt und ausgeführt werden kann.

NexusDS bietet zwei Erweiterungsmöglichkeiten: Eine Operator- und eine Diensterweiterung. Zur Operatorerweiterung setzt NexusDS auf ein flexibles Operatormodell, welches sich von bisherigen Ansätzen [2, 9, 127] durch die Beschreibung der Operatoren mit Hilfe von Metadaten unterscheidet. Dadurch kann ein Entwickler zusätzliche Operatoren in das System integrieren und dem System eine Beschreibung der Operatoren bieten. Die Metadatenbeschreibung enthält Angaben, wie beispielsweise die Anzahl der Ein- und Ausgänge, welche der Operator hat, welche Art von Daten an den jeweiligen Ein- und Ausgängen erwartet werden sowie Anforderungen an die Ausführungsumgebung, welche den Operator ausführen wird. Der Operator wird beim Operator-Repository (ORS) registriert und steht ab diesem Zeitpunkt zur Verfügung.

Das Dienstmodell erlaubt es zusätzliche Diensterweiterungen in das System einzubinden. Hierbei liegt dem Dienstmodell der gleiche Metadatenansatz wie beim Operatormodell zugrunde. Allerdings sehen die Metadaten anders aus und enthalten Informationen über das akzeptierte Nachrichtenformat, mit dem der Dienst angesprochen werden kann, über Abhängigkeiten zu weiteren Diensten im System oder Beschränkungen bezüglich der Ausführbarkeit des Dienstes. Die Dienste kommunizieren durch den Austausch von XML-Nachrichten und registrieren sich, wie Operatoren auch, durch Angabe der Metadaten bei einem Dienst-Repository und stehen dann zur Verfügung.

5 Sicherheitskonzept

Kapitel 5 geht auf die Fragestellung der Sicherheit in Datenstromverarbeitungssystemen ein. Um einen kontrollierten Zugriff auf die Daten sowie eine kontrollierte Verarbeitung zu gewährleisten, unterstützt NexusDS die Definition von Zugriffsrichtlinien, welche vor jeder Ausführung ausgewertet werden. Hierbei werden Richtlinien für *Datenzugriff*, *Datenverarbeitung* und *Datengranularität* unterschieden. Diese werden vom *Access Control Service* verwaltet. Verwandte Arbeiten finden sich im Datenbankbereich, bei dem die Zugriffsregelung wie in [74] beschrieben erfolgt. Zugriffsrechte werden hier für Rollen definiert, wobei ein bestimmter Benutzer ein Mitglied einer Rolle ist. Eine der ersten Arbeiten im Bereich der Datenstromverarbeitungssysteme wurde durch Lindner and Meier [88, 89] durchgeführt. Diese definieren einen allgemeinen Ansatz, wie sich ein Datenstromverarbeitungssystem sichern lässt. Die Idee sieht vor, nach beendeter Verarbeitung zu überprüfen, welche Datenelemente die definierten

Sicherheitsbedingungen nicht erfüllen und diese dann zu eliminieren. Wie ersichtlich ist, wird hierdurch unter Umständen viel unnötige Arbeit verrichtet. Andere Ansätze setzten bereits bei der Analyse der gestellten Anfragen ein und verfolgen das Umschreiben der Anfragen, um die Sicherheitsrichtlinien einzuhalten [29, 30]. Dadurch lassen sich unnötige Operationen vermeiden. Allerdings setzt das Konzept voraus, dass die Semantik der Operatoren bekannt ist, um die notwendigen Umschreibeoperationen vornehmen zu können. Nehme et al. [99] schlagen vor, die Zugriffsregelung durch die Einflechtung von Interpunktionen zu erreichen. Interpunktionen sind Datenelemente, die in den eigentlichen Datenstrom eingeflochten werden, um Metadaten zu transportieren. Hierzu werden in den eigentlichen Datenstrom noch Sicherheitsinterpunktionen eingeflochten, so dass zu jeder Zeit gewährleistet ist, dass die richtigen Sicherheitsrichtlinien vorliegen.

Zumeist konzentrieren sich die Konzepte auf zentralisierte Ansätze und bieten keine Möglichkeit, sensible Daten in verschiedenen Detailstufen zu filtern. Das Konzept in NexusDS schließt diese Lücken und vereinigt die Vorteile der vorgestellten Konzepte.

6 Deployment von Verarbeitungsgraphen

Kapitel 6 beschreibt ein flexibles Verfahren zum automatisierten Deployment von NPGM-Verarbeitungsgraphen. Bei der Entscheidung des Deployments der Verarbeitungsgraphen auf die zur Verfügung stehenden Rechenknoten berücksichtigt das Verfahren vor allem Anforderungen an die Dienstgüte- bzw. -qualität (sog. Quality of Service (QoS)-Anforderungen), welche zuvor von der Anwendung bzw. dem Anwendungsentwickler definiert wurden. Als Deployment wird hierbei die Fragmentierung des ursprünglichen Verarbeitungsgraphen sowie dem Deployment der einzelnen Fragmente auf Rechenknoten verstanden. Hierbei spielen die Kriterien, nach denen ein solches Deployment erfolgt, eine entscheidende Rolle, da es weitreichende Folgen für den weiteren Verlauf der Ausführung haben kann. Ein ungünstiges Deployment verringert die Qualität der Ausführung und kann zu einem hohen administrativen Aufwand, beispielsweise durch häufige Migrationsprozesse für das System führen. Hierbei wird die Annahme getroffen, dass der Anwendungsentwickler genau weiß, wie der für die Anwendung notwendige Rahmen aussieht. Beispielsweise könnte eine Anwendung existieren, welche die Bandbreitennutzung maximieren sowie die Latenz minimieren will. Schließlich sollten die eingesetzten Rechenknoten eine Verfügbarkeit von mindestens 75% aufweisen. Eine zweite Anwendung, die nun gleichzeitig das System benutzt, möchte die Bandbreitennutzung minimieren (weil sonst Kosten anfallen, da ihre Ausführung beispielsweise einem anderem Lizenzmodell unterliegt). Dieser Anwendung reicht bzgl. der eingesetzten Rechenknoten eine Verfügbarkeit von 50% aus. Dieses einfache Beispiel zeigt, dass Anwendungen unterschiedliche QoS-Anforderungen an die Ausführung und an das Deployment stellen. Die unterschiedlichen QoS-Anforderungen der Anwendungen, möglicherweise verschiedener Domänen, können auch im Konflikt zueinander stehen. Diese Tatsache macht einen anwendungsspezifischen

Deployment-Mechanismus erforderlich, welcher sich an die QoS-Anforderungen der jeweiligen Anwendung anpassen lässt.

Multi-target operator placement (M-TOP) berücksichtigt die von der Anwendung definierten QoS-Anforderungen und überträgt diese auf den Deployment-Vorgang. Damit löst M-TOP das Deployment-Problem. Das geschieht durch einen mehrstufigen Ansatz, welcher zum Ziel hat, potentiell ungültige Deployment-Entscheidungen frühzeitig auszuschließen und so den entsprechenden Suchraum bald zu reduzieren. In der Literatur gibt es für das Deployment eine Reihe von Vorschläge. So schlägt Zhou et al. [153] vor, das initiale Deployment durch Reduzierung der Latenz und die Anpassung im laufenden Betrieb durch die Bestimmung der Systemauslastung jedes einzelnen Knotens vorzunehmen. Amini et al. [10] hingegen schlägt vor, das Deployment abhängig von der Priorität der jeweiligen Operatoren vorzunehmen. Wichtigere Operatoren werden hierbei bevorzugt behandelt und bekommen entsprechend mehr Ressourcen zugewiesen.

M-TOP findet ein Deployment für einen Verarbeitungsgraphen in heterogenen Umgebungen, sprich Umgebungen mit Rechenknoten, welche unterschiedliche Eigenschaften aufweisen wie beispielsweise das Vorhandensein einer GPU. M-TOP berechnet hierfür den Nutzwert für ein gegebenes Szenario, verwendet eine Kombination aus Worst-Fit- und Best-Fit-Methode und liefert eine Lösung unter Einhaltung gegebener QoS-Anforderungen. Hierbei bestehen QoS-Anforderungen aus einer *Engpassbedingung* (Bottleneck Condition), einer *relativen Wichtigkeit* (Relative Importance) und einem *Ordnungsschema* (Rank Scheme). M-TOP besteht aus sechs Verarbeitungsschritten, namentlich *Vereinigung* (Conflation), *früher Ausschluss* (Early Prune), *Graphmontage* (Graph Assembly), *Rangfolge* (Ranking), *Abbildung* (Mapping) und schließlich *Ausführung* (Execution). *Vereinigung* schließt Operatoren des Verarbeitungsgraphen zusammen, die gemeinsam auf denselben Rechenknoten abgebildet werden sollen. Dieser Vorgang resultiert in einem virtuellen Operator und ist eine Kombination aus den Anforderungen der Vereinigung der Operatoren, was insbesondere bedeutet, dass ein Rechenknoten die Anforderungen aller enthaltenen Operatoren erfüllen muss. Die Operator-Anforderungen ergeben sich aus den jeweiligen Metadaten. *Früher Ausschluss* bestimmt dann potentielle Kandidatenknoten für das Deployment der Operatoren und erstellt pro Operator eine entsprechende Kandidatenknotenliste. Im Anschluss wird im Schritt *Graphmontage* ein Kandidateknotennetzwerk konstruiert, indem Kandidatenknoten entfernt werden, welche die gegebenen Engpassbedingung nicht einhalten und dann passende Verbindungen zu den anderen Kandidatenknoten gesucht. Die Kandidatenknoten und deren Verbindungen werden anschließend durch *Rangfolge* bewertet und in eine Reihenfolge gebracht. Im letzten Schritt vor der *Ausführung* sucht der Schritt *Abbildung* nach einer passenden Abbildung von (virtuellen) Operatoren zu Kandidatenknoten. Da dieser Schritt NP-vollständig ist, muss hier ein Näherungsverfahren angewendet werden. Unter Verwendung der bisher durchgeführten Schritte erfolgt die Abbildung durch Anwendung eines *genetischen Algorithmus* [95]. Dieses Lösungsverfahren ist der Evolutionstheorie entlehnt. Hierbei wird eine Lösung als *Chromosom* kodiert. Darauf werden die primitiven Operationen *Mutation* und *Rekombination* angewandt. Daraus ergeben sich neue

Lösungen, aus denen zu einem bestimmten Zeitpunkt die gerade (unter den gegebenen Voraussetzungen) beste Lösung bzw. Chromosom herausgegriffen wird.

7 Werkzeugunterstützung

Kapitel 7 beschäftigt sich mit der Frage nach einer passenden Werkzeugunterstützung für ein Datenstromverarbeitungssystem.

Die vorangegangenen Kapitel haben gezeigt, dass ein flexibles und erweiterbares Datenstromverarbeitungssystem Vorteile bringt und wie ein solches System aussehen kann. Damit ein solches System tatsächlich verwendbar wird, bedarf es einer geeigneten Werkzeugunterstützung. Diese sollte einmal die Modellierung des technischen Kontextes der einzelnen Komponenten des Systems (Operator- und Dienst-Metadaten) unterstützen und auch die Formulierung entsprechender Verarbeitungsgraphen. Zu diesem Zweck wurde der *NexusDSEditor* entwickelt. Hierbei handelt es sich um einen grafischen Editor, mit dem die Anwendungsentwicklung für Datenstromverarbeitungssysteme unterstützt werden kann. Aufgrund der Komplexität des NexusDS-Systems und der damit zusammenhängenden hohen Anforderungen, welche an einen Anwendungsentwickler gesetzt werden, eignet sich ein solches Werkzeug zur Reduktion der Fehlerwahrscheinlichkeit durch geeignete Unterstützung des Entwicklungsprozesses.

Es gibt eine Vielzahl an Werkzeugen, womit sich die einzelnen Prozesse, welche zur Erstellung einer Anwendung notwendig sind, abbilden lassen. Jedoch gibt es das prinzipielle Problem, dass viele weitere Werkzeuge erlernt und angewendet werden müssen, um eine entsprechende Datenstromanwendung zu realisieren. Einen Überblick vorhandener Werkzeuge für die Modellierung und Verwaltung von Kontextdaten wird durch Bettini et al. [24] gegeben. Werkzeuge umfassen die Bearbeitung und Erstellung von Extensible Markup Language (XML)-Dateien durch Werkzeuge wie *XMLSpy*[1], die Erstellung von Unified Modeling Language (UML)-Dateien durch Werkzeuge wie *IBM Rational Rose*[2] oder das Testen des Systems durch das Erstellen und Versenden von Simple Object Access Protocol (SOAP)-Nachrichten mit Werkzeugen wie *soapUI*[3]. Alle genannten Ansätze sind generische Ansätze, und lassen sich in dem in dieser Arbeit formulierten Kontext anwenden, jedoch sind diese Werkzeuge auch aufgrund ihrer Allgemeingültigkeit nicht ideal einzusetzen.

Der NexusDSEditor vereinheitlicht Anwendungsentwicklungs- sowie Überwachungsprozesse, um Entwickler bei der Erstellung einer Erweiterung für das Datenstromverarbeitungssystem oder bei der Erstellung eines Verarbeitungsgraphen zu unterstützen. Allgemein bildet der NexusDSEditor die Schnittstelle zwischen dem Datenstromverarbeitungssystem und einem Benutzer, zumeist in Gestalt eines Anwendungsentwicklers. Der NexusDSEditor leistet Folgendes: Er ermöglicht die *Erweiterung des Nexus-Umgebungsmodells*. Aufgrund des zugrundeliegenden Schemas lässt der NexusDSEditor nur eine mit dem *Schema konforme Modellierung* zu,

[1] **http://www.altova.com/products/xmlspy/xml_editor.html**
[2] **http://www.ibm.com/software/awdtools/developer/rose/**
[3] **http://www.soapui.org/**

er bietet eine Schnittstelle zur *Kommunikation mit den Context-Servern des Nexus-Systems*, er ermöglicht das *Visualisieren von Umgebungsmodelldaten* in einem internen oder externen Betrachter (wie beispielsweise *GoogleEarth*[4]), er vereinfacht das *Schnüren der Operator- und Dienstpaketen*, welche dann im jeweiligen Repository abgelegt werden können und er ermöglicht die *Modellierung von Verarbeitungsgraphen* sowie die *Festlegung der zugehörigen Deployment-Anforderungen*.

8 Schlussfolgerungen und Ausblick

Kapitel 8 schließt diese Arbeit mit einem Fazit ab und bietet einen Ausblick auf mögliche zukünftige Arbeiten.

Ausgehend von typischen Anwendungsszenarien kontextbezogener Anwenungen wurden Anforderungen definiert. Diese Anwendungen stellen besondere Anforderungen und stellen wichtige Kriterien dar, die auf geeignete Weise in Datenstromverarbeitungssystemen unterstützt werden müssen. Für die definierten Anforderungen wurden in einem zweiten Schritt Methoden und Konzepte entwickelt, welche es einem Datenstromverarbeitungssystem erlaubt sich auf die Bedürfnisse kontextbezogener Anwendungen anzupassen und somit eine flexible Verarbeitung der damit zusammenhängenden Kontextdatenströme zu ermöglichen. Es entstand das flexible und erweiterbare Datenstromverarbeitungssystem NexusDS, das besonders auf Bedürfnisse kontextbezogener Anwendungen zugeschnitten ist. Ein besonderes Merkmal ist das flexible Operatormodell, um neue Funktionalität zur Datenverarbeitung in das System integrieren zu können. Zudem kann NexusDS durch die Integration zusätzlicher Dienste funktional erweitert werden.

Da NexusDS ein verteiltes Datenstromverarbeitungssystem ist und die Verarbeitung der Daten Restriktionen, wie einer Beschränkung der Verarbeitung auf bestimmte Rechenknoten unterliegen kann, bietet NexusDS die Möglichkeit, das Deployment den Anwendungsbedürfnissen anzupassen. Insbesondere bietet es auch die Möglichkeit, das Deployment anhand von QoS-Anforderungen anzupassen. Hierzu wird der Verarbeitungsgraph mit entsprechenden Laufzeit- und Deployment-Restriktionen annotiert. Beispielsweise lässt sich durch diesen Mechanismus die Ausführung bestimmter Teile des Anfragegraphen auf eine bestimmte administrative Domäne beschränken. Für das automatische Deployment wurde M-TOP vorgestellt, ein multikriterielles Verteilungsverfahren, das QoS-Anforderungen sowie die im Verarbeitungsgraphen definierten Restriktionen geeignet unterstützt und berücksichtigt. M-TOP definiert hierfür Vorverarbeitungsschritte, um die entsprechenden Restriktionen auf Verarbeitungsgraphebene zu berücksichtigen. Die Abbildung auf die zur Verfügung stehende Infrastruktur erfolgt dann unter Anwendung eines Genetischen Algorithmus' [95]. Dieser wird durch die vorangegangenen Vorverarbeitungsschritte geeignet unterstützt und erreicht gute Platzierungsergebnisse bei gleichzeitiger Reduzierung der Zeit zur Lösungsfindung. Da allerdings auch der Datenzugriff geregelt werden muss, müssen geeignete Zugriffsmechanismen vorhanden sein, um einen

[4] **http://www.google.com/earth/index.html/**

kontrollierten Zugriff und eine kontrollierte Verarbeitung der Daten zu gewährleisten. Hierfür bietet NexusDS einen Zugriffskontrolldienst, durch den der Zugriff auf Daten und die Verarbeitung geregelt werden kann. So lassen sich beispielsweise für bestimmte Anwender Positionsinformationen unschärfer machen oder ein Zugriff komplett verwehren, um einen Missbrauch der Daten zu verhindern.

Abgeschlossen wird die Arbeit durch die Vorstellung der im Rahmen dieser Arbeit entstandenen Werkzeugunterstützung für Datenstromverarbeitungssysteme. Der NexusDSEditor unterstützt sowohl die Modellierung des technischen Kontextes der einzelnen Systemkomponenten (Operator- und Dienst-Metadaten) als auch die Formulierung entsprechender Verarbeitungsgraphen.

Part I

Motivation, Requirements and Foundation

1 Introduction

1.1 Motivation

The amount of data we have to face every day grows steadily. The velocity in which new data is produced increases every day as well. Eric Schmidt, CEO of Google, once estimated the size of the World Wide Web (WWW) is about 5 EBs (10^{18} Bytes = 1.000.000.000.000.000.000 Bytes) [27]. This amount of data covers only the WWW-related data—also non-WWW-related data is being produced and the total amount grows every day. In 2009, Andreas Weigend, former chief scientist of Amazon, predicted "*that human beings would generate more data in 2009 than in all prior human history*" [135]. According to Eric Schmidt in 2011 we are adding this amount of data to the *human Database* (human DB) every two days. This human DB is composed of many different data sources, including the WWW with all its publicly available unstructured data and structured data, rather private structured data present in scientific or medical DBs as well as sensors producing a continuous stream of data, e. g. to sense the environment. This can be summed up by the term *big data. Big data* characterizes the acknowledgment of "*the exponential growth, availability and use of information in the data-rich landscape of tomorrow*" [56]. According to Gartner, solving the big data challenge involves more than just managing high volumes of data [56]. This becomes evident if we look at the huge number of different sources, data is available from: The public data present in the WWW, the rather private scientific data stored in DBs, and the streams of highly volatile and time-critical data. Thus, not only the *volume* of data should be considered but also the *variety* and *velocity* of data must be taken into account in order to keep track with the highly dynamic and manifold nature of data [56].

While research proposed many interesting and efficient solutions to process unstructured as well as structured data of static nature, for the dynamic nature of streamed data there are still open questions w.r.t. the big data issue for context-aware applications. This especially includes

providing concepts and coping with the integration of highly domain-specific functionality for applications relying on the data stream processing paradigm. The term *static* here means that the update frequency is low compared to data streams which we define as being *dynamic* since updates most likely occur with high frequency. A data stream is characterized as a potentially infinite flow of data elements from one or more data sources. The processing of such data streams is typically done in two steps: Data elements are collected from the data sources and are processed according to a processing definition consisting of a defined set and ordering of operators, which are interconnected defining a processing pipeline. Over the past few years data stream processing has been in the focus of research all over the world. Research ranges from proposals for Data Stream Processing System (DSPS) architectures [2, 14, 57] over adjusted stream processing techniques [112, 126, 153] to query distribution and re-use resulting in more and more sophisticated techniques [10, 153].

The term DSPS will be used as a synonym for Data Stream Management System (DSMS) throughout this thesis. Centralized approaches such as [14] exist also, but this thesis will not further differentiate between the two DSPS variants but will assume that DSPSs are distributed as this variant constitutes the current state-of-the-art.

This thesis covers the Database Management System (DBMS)-oriented perspective on data stream processing. In the DB context this was firstly mentioned by Babcock et al. [16]. They discuss the question of how a management system for the domain of data stream processing, i. e. how a DSPS should look like. Conversely, this means to transfer the management functionalities of DBMSs and to adapt them to meet the specific requirements of DSPS. However, DSPSs never reached such a huge community as DBMSs did. Depending on the domain of interest, e. g. context-aware visualization, the processing of such data is often related to highly domain-specific functionality. This domain-specific functionality is—beside others—specified in terms of highly specialized operators that may require specialized hardware to run smoothly. E. g. in the context of visualization an operator that renders a scenery requiring a Graphics Processing Unit (GPU). The seamless integration of these highly specialized operators into DSPSs is a key feature to address and adequately support a wide range of applications relying on the data stream processing paradigm. This is because the potential infiniteness of data streams prohibit their storage and postponed processing. Also data transfers must be reduced to a minimum to permit an efficient processing of data streams since high data volumes are assumed. This creates a strong dependency between application requirements on the one side and system capabilities on the other side. This fact must be taken into account by DSPSs [43].

It could be reasonably contended that the development of this application-specific functionality has to be done anyway in order to make the application finally work. However, as described in [43] an adaptation problem is still persisting. Usually, DSPSs provide a generic querying and processing mechanism to process the streamed data in an application-independent manner. This especially means that operators are rather generic and resemble those of DBMSs. But context-aware applications rely on models of the physical world which often have different data formats. This model of the physical world is given by static context information such as map data and 3D models as well as dynamic information from billions of sensors located

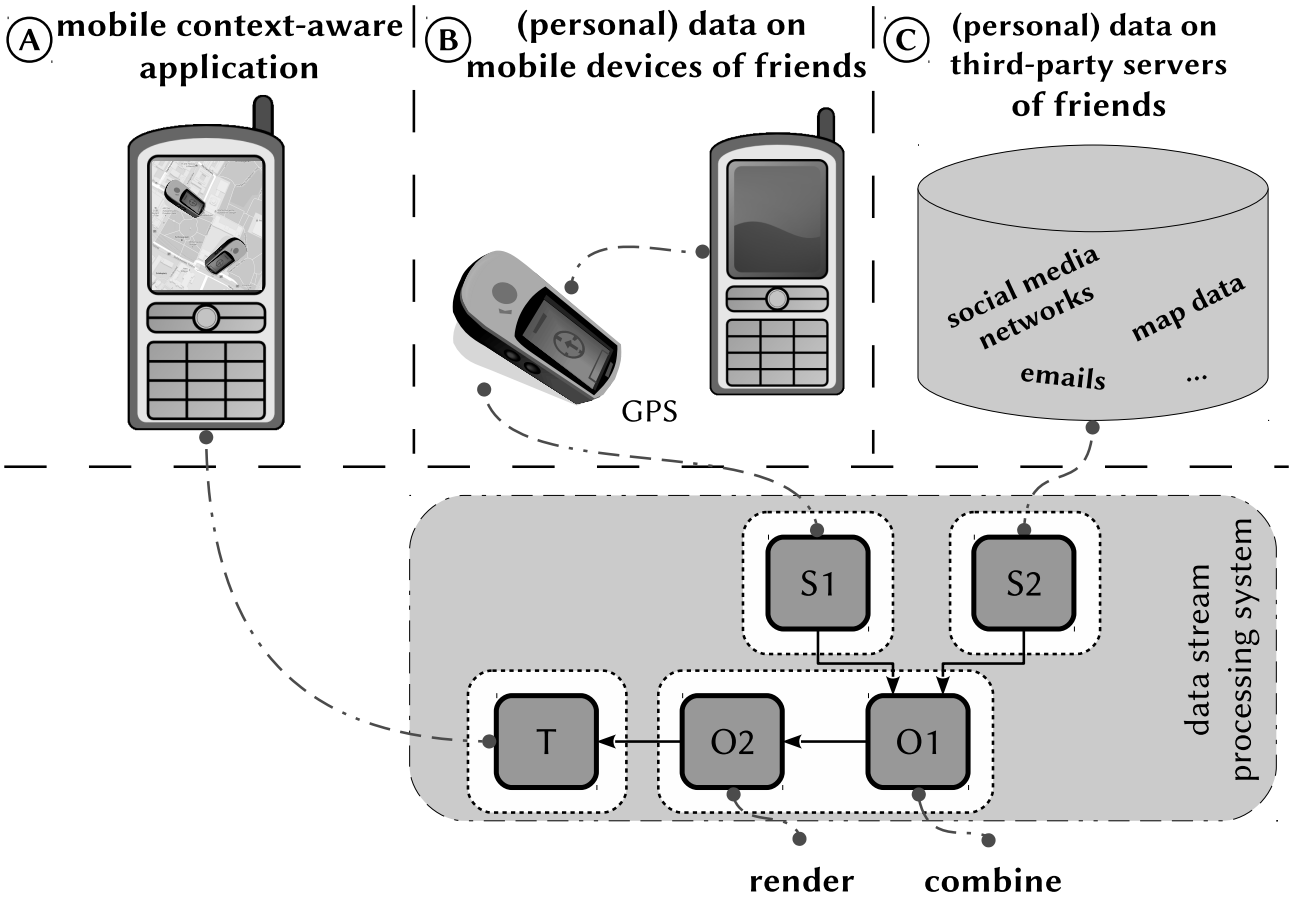


Figure 1.1: Application scenario of a mobile context-aware application tracing friends.

in our physical environment, e. g. Global Positioning System (GPS) sensors in mobile devices. They also heavily rely on highly specialized data operations, as discussed in Chapter 2. Sensors and more general data sources, such as position data of moving objects, produce streamed data continuously that are consumed by context-aware applications. A prominent example of context-aware applications are location-aware applications which rely on the context information regarding their surroundings with respect to the current position. Due to these reasons, the model of providing rather fixed querying and processing mechanisms does not hold for the domain of stream processing as regards context-aware applications.

As depicted in Figure 1.1, context-aware applications may produce data streams (denoted by Ⓑ) and at the same time consume data streams (denoted by Ⓐ). In this scenario, a user running a *mobile context-aware application* Ⓐ wants to visualize a map of its surroundings. The map displays his friends pinned to their current location. The friends have *mobile devices* Ⓑ with numerous integrated sensors which allow to sense the environment. E. g. they provide their current position originating from a GPS sensor producing a continuous stream of position data elements. However, to get a nicely displayed map of its surrounding the mobile application also needs additional data originating from *third-party servers* Ⓒ located in the WWW. These could be servers providing map data or personal data of the friends of interest. The mobile context-aware application is designed to receive the resulting image of the sur-

rounding map. Therefore, the position data originating from the mobile devices GPS sensor (denoted by the source operator *S1*) and the personal data originating from third-party servers (denoted by the source operator *S2*) must be processed according to a stream processing graph (SP graph) which is executed within a DSPS. A SP graph is usually not processed on a single machine but distributed on different machines. These are denoted by the different dashed white boxes. A SP graph consists of a number of sources **S**, operators **O**, and sinks **T** that are interconnected, building a network of operators[1]. Thereby, the sources **S1** and **S2** provide the data streams for the operators **O1** and **O2**. **O1** *combines* the data originating from the two sources. Therefore, each mobile device (representing a friend) is connected to a social media profile on third-party servers. In order to augment the position of a mobile device with additional data, for each device the respective data must be extracted from third-party servers. **O2** receives the preprocessed data and *renders* an image of the scenery. This operator heavily relies on specialized hardware, i. e. a GPU, thus making a seamless and flexible integration of specialized operators mandatory. The result is send to the sink **T**, representing the mobile context-aware application.

Knowing the bandwidth requirements of this application, an application developer can clearly identify the specific QoS requirements for the correct distribution of the SP graph. These requirements are a good indicator for the DSPS to decide how best to distribute the SP graph to meet the application requirements. It is important to note that within the same DSPS many other different applications might exist. These applications might have different requirements. E. g. an interactive stream-based game application in a first place needs a fast and reactive SP graph, i. e. it needs the latency to be minimized.

Moreover, users participating in the process might not want to expose their current location to potentially unknown parties, restricting e. g. data access to known or trusted ones only. Therefore, additionally to the flexible integration of specialized operators security aspects must also be considered, limiting the access of data as well as the granularity at which data is made available. In the sample scenario, the resulting data for the third-party server might be less accurate, thus only indicating in which state a user currently is obfuscating his or her actual position.

In the scenario sketched here it seems reasonable not to perform the combination and rendering of the data on the mobile device. This is not recommendable, since potential high volume data transfers over mobile networks is problematic. Moreover, processing power of mobile devices is often insufficient. Thus, these operations should be performed near to the original data and furthermore in an adequate environment, i. e. DSPSs. But many DSPSs lack the integration of specialized operators. Besides, the particular requirement characteristics of the operators are not considered as well.

To avoid context-aware applications consisting of many isolated application parts ”knitted together in a hurry”, DSPSs should provide an adequate integration mechanism for such applications. This reduces redundancy and increases reuse of existing functionality. That is, for

[1]Sources and sinks are special operators. Sources exclusively produce data whereas sinks exclusively consume data.

both economic as well as technical reasons, it is beneficial to provide mechanisms to integrate such highly domain-specific functionality in DSPSs and exploit—whenever possible—already existing functionality. Also, the security aspects in such an extensible distributed environment are not fully considered, as will be presented further on. Finally, an adequate distribution strategy needs to be implemented to reflect the specific QoS requirements.

This thesis presents concepts to enable a flexible and extensible way to process streams of context data. These concepts have been integrated into a flexible and distributed stream processing system called NexusDS. NexusDS is especially tailored to applications of the context-aware domain. The idea of NexusDS as well as the related concepts we have developed are presented and discussed throughout this thesis.

1.2 Contributions and Outline

The contributions of this thesis can be classified in three main topics: A data stream processing system for context-aware applications, dedicated mechanisms for context-aware application integration, and tool support for these environments. The following elementary contributions derive from these topics:

1. Chapter 2 provides an *analysis* of representative context-aware applications and extraction of their requirements. The requirements formulate the motivation and the necessity for a flexible and extensible DSPS.
2. In Chapter 3 an *architecture* is proposed of a flexible and extensible DSPS which targets the necessary requirements from Chapter 2 is presented. A DSPS based on this architecture can be extended by integrating additional operators responsible for data processing and services realizing additional interaction patterns with context-aware applications.
3. Besides the integration of custom functionality to suitably support context-aware applications, a mechanism to express their special *requirements*, e. g. constrain the execution of operators to only certain processing nodes, is a key to success. Therefore, a SP graph model has been developed which reflects these constraints by allowing to annotate the graph by constraints. These constraints span a constraint model. In Chapter 4, the constraint model is presented, and a SP graph model that reflects formulated constraints.
4. NexusDS provides support for *security* related issues. Chapter 5 presents how security related constraints are supported at a SP graph level. In short, this happens by augmenting the SP graph by the relevant security policies. Three different types of such policies are supported: Access Control (AC) controlling data access, Process Control (PC) influencing how data is processed, and Granularity Control (GC) defining the Level of Detail (LOD) the data is made available.
5. Once the additional functionality is integrated and the corresponding constraint-enriched SP graph is formulated, the DSPS has to find a suitable deployment, i. e. a suitable distribution of the SP graph which places the operators, in order to satisfy the

predefined QoS requirements. Thereby, through the placement algorithm constraints as well as QoS targets are considered. This is discussed in Chapter 6.

6. An appropriate design and feature definition for tool support for such complex systems is essential. Chapter 7 introduces tool support for the modeling of context-aware applications.

The combination of all contributions enables the realization of a DSPS which targets the requirements of context-aware applications. The details are presented and discussed in the next chapters.

2 Requirements and Foundations

In this chapter the requirement formulation and foundations for the remaining chapters are provided. First, in Section 2.1 the problem of DSPS adaptability is introduced and briefly discussed. Then, in Section 2.2, the term *context-aware application* is defined. Motivating example scenarios from the domain of context-aware applications are presented in Section 2.3. These application examples are analyzed, and resulting requirements are discussed in Section 2.4. The necessary background for the rest of this thesis is given in Section 2.5. In that section, first the Nexus project is presented, providing an overview picture this thesis is embedded in. Afterwards, the required foundation is given. Here, a technological and a fundamental introduction to related work to this thesis is provided. Chapter 2 concludes with a short summary.

2.1 Problem Statement

Nowadays, many sensors produce a huge amount of data that is time-constrained and should thus be processed near creation time. One reason for this is that sensor data results in potentially unbound streams. This fact makes the storing of data streams and its postponed processing practically impossible. Another reason is that this data has stringent time constraints and must therefore be processed near creation time.

In the past decade many studies have been conducted in the field of data stream processing systems. The proposals ranged from data stream processing system architectures such as [2] to sophisticated processing techniques [129]. Nowadays, DSPS are state-of-the-art since they scale well with increasing workload and thus enable an efficient processing of data sources producing streams of data in a distributed environment. These systems often provide a declarative query language and allow to continuously process incoming data streams in a distributed fashion [15]. The query is first mapped to an SP graph containing operators. In a second step

Figure 2.1: Real-time Flow Visualization of Stream Ribbons which shows the airflow in a building based on the user's current position. [42]

the SP graph is distributed across available computing nodes. Some systems provide a way to directly define the data processing by drawing a corresponding stream processing graph (SP graph) [3] or by providing a programming model allowing to script the way the data streams are processed [57].

Nevertheless, none of the systems have reached a general acceptability for a huge number of applications and application domains, such as DBMSs did. This is due to the fact that they do not consider the specific needs of real-time applications. Depending on the domain of interest, e. g. visualization, the processing of such data often depends on highly domain-specific functionality but at the same time relies on common functionality. Specific functionality for this domain of interest is, e. g. the rendering of a scene which is usually not part of the common functionality of a DSPS such as predicate-based filtering of data. For instance, an application which visualizes stream-based data has stringent timing constraints or may need a specific hardware environment to smoothly process the data. Such a complex application is depicted in Figure 2.1. In this example, the air flow in a room is simulated and visualized. The air flow also adapts to changing positions of objects moving in the room. The processing of such a scenario is a highly complex task and cannot be reasonably performed on mobile devices. Thus, specialized hardware as well as remote processing of complex tasks must be exploited.

Furthermore, users may want to add additional constraints to the actual execution. For example, for security reasons they may want to restrict the set of nodes that participates in data processing. This security restriction may originate even from the system itself, if we think of a setting in which the system is part of a production environment in factories where sensitive data should not cross certain boundaries. In short, as explained in [43], this results in a permanent adaptation necessity in today's DSPS to new requirements of domains such as visualization. Finally, this means that each application domain has its specific processing functionalities although they rely on common processing techniques, i. e. data stream processing.

We argue that many applications, although originating from different application domains mostly share common processing principles. This calls for a data stream processing concept that allows to express the particular characteristics and requirements of each considered application. This remarkably reduces development overhead and enhances infrastructure exploitation as well as overall performance.

2.2 Context-aware Applications

What exactly are context-aware applications? Context-aware applications represent a class of applications that adapt their behavior and functional range according to contextual information of their usually near surroundings to increase effectiveness and usability. The environmental adaptation is done without the explicit user intervention. An example for a classical context-aware application is a navigation application. By the movement of the user the navigation application adapts its display according to the current user's position. The application also checks if the user is still on track. If not, depending on the situation, the application needs to display an adequate warning.

In general, *context information* strongly depends on the particular application. For the navigation application, the user's position is the context information of interest. In contrast for a mobile phone application, for adapting the display brightness to its surrounding the current light intensity is the context information of interest. The emotional state of the user might also be important for the context-aware application. According to Dey and Abowd [51], we define context information as follows:

> *"Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves. [...] A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task."*

This flexibility in defining what exactly context information means for a certain application makes an adequate integration into existing systems difficult and represents a great challenge.

This is because the application's concrete functional and operational requirements heavily depend on the domain of interest and thus on the concrete interpretation of contextual information. Context data is very useful in mobile and dynamic environments, posing many challenges, e. g. how to find out which context data is actually relevant for the application, or how to handle continuous updates of context data in such environments. These aspects are discussed by the sample scenarios in Section 2.3 and the resulting requirements in Section 2.4.

In our application scenarios, many applications share a common basis to cope with the dynamic nature of context data streams: Data stream processing. Thus, using DSPS seams reasonable while integrating highly specialized components to reflect each application's requirements. Thereby the motivation is to reuse existing functionality already present in an existing DSPS and extend the system by the missing parts, as needed by the respective domains of interest. This way a tight integration of the application in terms of data processing is achieved. This means, as described in [43], there is a continuous need for such systems to adapt to applications.

2.3 Example Scenarios

For clarification, the following four non-trivial applications were considered: **location-aware visualization pipeline** in Section 2.3.1, **management support in smart factories** in Section 2.3.2, **storing moving traces of moving objects** in Section 2.3.3, and **location-based service application** in Section 2.3.4. When presenting a scenario via figure, solid directed arrows indicate the streamed data which is forwarded by each operator. Dashed directed arrows indicate parameter updates, i. e. changes in the parametrization of operators.

2.3.1 Location-aware Visualization Pipeline

A complex data stream scenario that goes beyond the current state-of-the-art is an interactive and location-aware visualization application as depicted in Figure 2.2 (the resulting visualization is shown in Figure 2.1). In this example, the air flow in a room is simulated and visualized. Objects moving in the room are tracked by the *Position Tracker* whereas the status of the windows is tracked by *Window Tracker*. The *Fluid Solver* simulates the velocity field, which depends on the objects tracked. The *Environment* source delivers the static model data of the environment (in our case the buildings) that are to be displayed. The *Calculate Stream Lines* operator generates and calculates streamlines based on the velocity field. To visualize the twist induced by the velocity field, stream ribbons are calculated from the streamlines by the *Calculate Stream Ribbons* operator. The geometry obtained is rendered by the *Rendering* operator, which produces image output. The rendering is usually not performed on the mobile client but on a remote server. The main reasons for this are the energy constraints of mobile devices and their typically insufficient processing power. This image output can be displayed on desktop

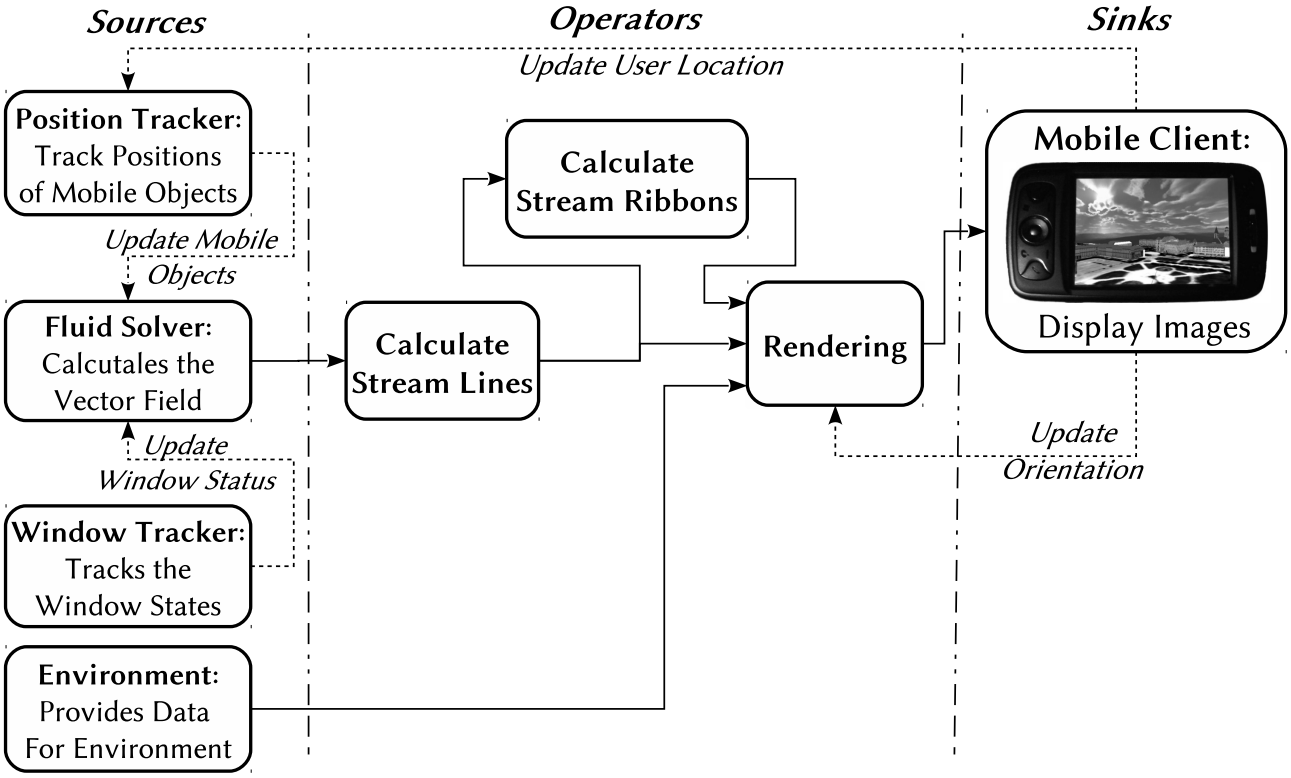


Figure 2.2: Visualization scenario realizing a flow visualization of airflows in buildings.

computers but can also be sent to *Mobile Clients* which do not have the capabilities to render complex scenarios. Preferably, the rendering step is executed on specialized graphics hardware (GPU) to get even better performance. User interaction, such as rotating and panning the scene, can be forwarded to the operators via parameter updates.

Thus, in order for the DSPS to support the location-aware visualization pipeline, it needs adequate integration mechanisms for its domain-specific operators and the respective constraints. E. g. for the efficient execution of the domain-specific Rendering operator, the presence of a GPU is mandatory. Furthermore, different interaction patterns must be integrated, e. g. changing the viewport for subsequent rendering steps and arbitrary data should be processable, ranging from structured data to unstructured and custom binary data as the rendered images.

2.3.2 Management Support in Smart Factories

A lot of influencing factors can cause disturbances in production processes of today's smart factories [91]. Unsteadiness of the demand for a product, changes of orders of a customer, the delayed delivery of raw materials, failures of machines or the decreasing quality of the products require a quick adaptation of the production process. In order to perform such quick adaptation, it is necessary that the responsible persons, e. g. production managers or maintenance staff, can get information on the current state of the production facilities, failures or actions that are required at any time. Another topic of interest is the failure management

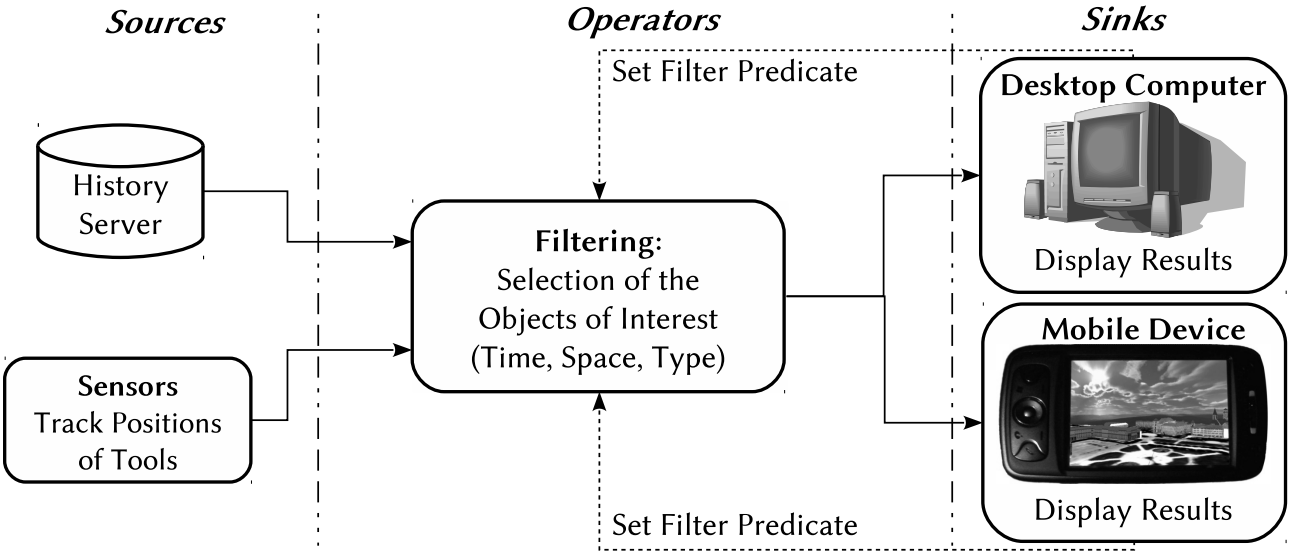


Figure 2.3: Management support in smart factories.

in such an environment. If a failure occurs during production, it is very important for staff members to get informed instantly of the problem, locate it fast, and get appropriate failure information. Thereby the responsible persons interact with the DSPS via interfaces. The interfaces constitute sinks for data streams in a DSPS. Selections based on time, type and location can be implemented as a selection operator in a DSPS, thus *Filtering* data elements as they arrive, as shown in Figure 2.3, exploiting existing functionality. In such scenarios, the data stream system can also be used to propagate measured values from the sensors to a history server, making the data available for retrospective failure analysis. This step is not shown in Figure 2.3 but is comparable to the one depicted in Figure 2.4.

Data originating from production facilities often are a business secret of the company owning the factory. Thus it is important that applications or application developers can restrict the set of nodes for data processing to nodes owned or controlled by the company or at least to nodes the company trusts. Furthermore, the company is interested in restricting and controlling the actual access to this data only to individuals who are entitled. This means, data processing and access should satisfy certain policies, which are defined by the respective data owners or are even established in law.

2.3.3 Storing Moving Object Traces

With the increasing use of sensor technology, the compression of sensor data streams is getting more and more important to reduce both the costs of further processing as well as the data volume for persistent storage. An example scenario is depicted in Figure 2.4. A GPS receiver (*Mobile Device with GPS Sensor*) produces a stream of position updates, which is first reduced to positions within a given area (*Selection* operator). The resulting stream is partitioned into windows by the *Partition* operator which forms the input for the *Compression*

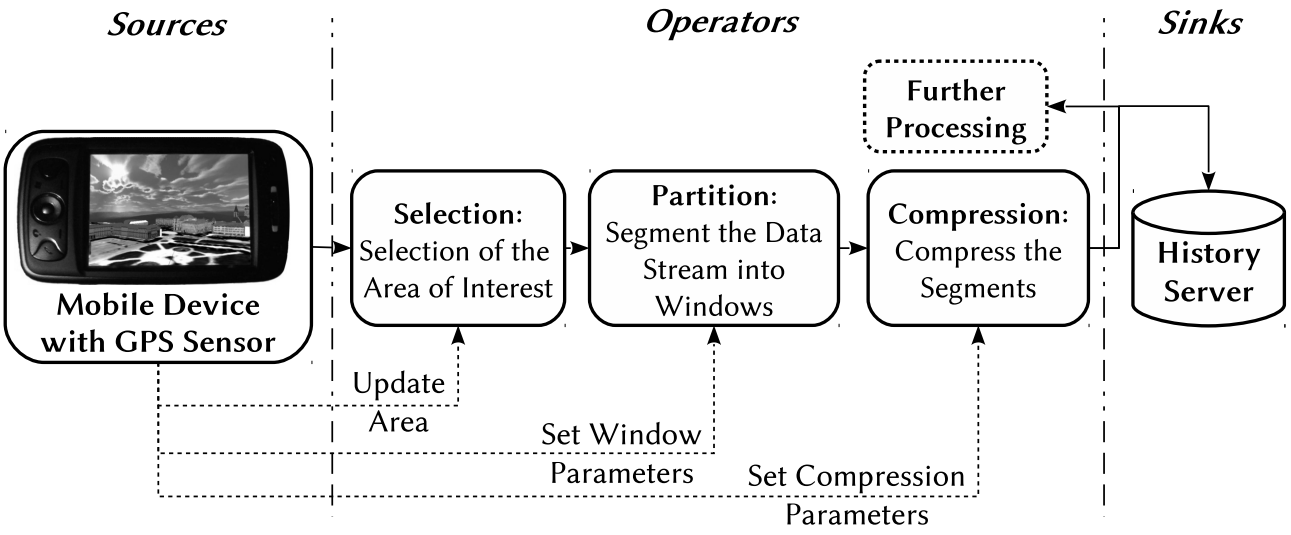


Figure 2.4: Storing of moving objects' traces.

operator. The compressed position information can be stored in a position *History Server* for later analysis or can be further processed by subsequent operators (denoted by the box *Further Processing*).

This approach allows moving the generic part of the functionality out of the compression operator and implementing it as an additional operator, i. e. *Selection* and *Window*. This increases the performance, as the different operators can be deployed on different nodes and executed in parallel. Furthermore, this enhances reusability of the generic parts [70]. This way, on one side new applications benefit from existing functionality and on the other side future ones will also benefit from functionality integrated before. The DSPS grows with the application requirements. An additional incentive for integration is the possibility of distributing the actual processing arbitrarily across participating nodes.

2.3.4 Location-based Service Application

The proliferation of mobile devices has favored the development and provisioning of so-called location based service (LBS). LBS offer a higher service such as a navigation service of an information portal depending on the user's current position. In this scenario, we describe a LBS named *Squebber*. Squebber manages virtual messages which have been placed by users. The messages can have an arbitrary content such as an invitation to some happening nearby. Squebber users have a user space which stores personal information, such as the favorite music band or favorite food. Squebber reveals potential friendship among users according to common interests. The Squebber scenario is depicted in Figure 2.5. Squebber users can add messages for their friends and attach them to a certain position, e. g. at the user's favorite spot. These messages are updated in the *Squebber Source*. Users can set the privacy of their position information and set the *Filtering* accordingly. Friends are allowed to see exactly where a

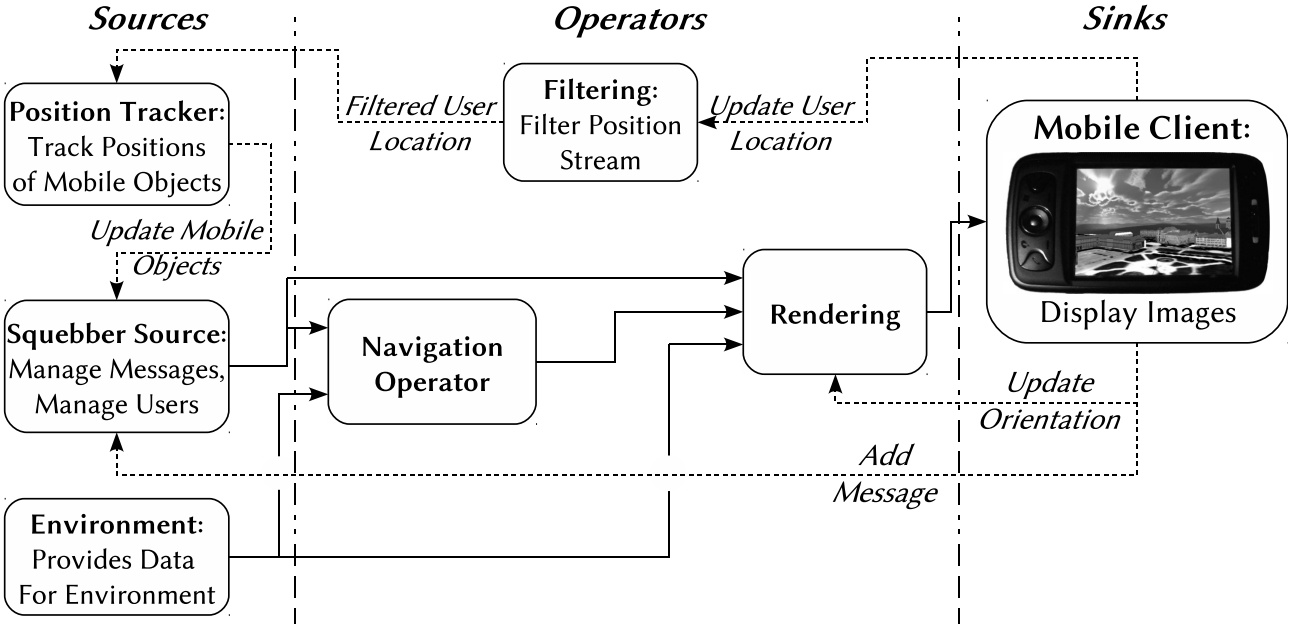


Figure 2.5: Location-based service scenario realizing the notification and navigation system 'Squebber'.

certain user is, facilitating spontaneous meetings. Other users might only be allowed to know the administrative district a certain user currently is in.

Once a user enters a certain area nearby some message which has been placed by his/her friend, a notification with the message content pops up in the mobile device. Such a notification might be an invitation to a birthday party. The friend now may accept the invitation and want to know how to get there. For this purpose Squebber creates a dedicated *Navigation* for that friend and gets him to the place where the birthday party is using *environmental data*, which is *rendered* and then sent to the *mobile device*.

This scenario illustrates that data providers need to define fine-grained access to data. Authentication of users is necessary to effectively restrict the access to data. Thereby, the user wants to adjust the quality of data he owns, e. g. position data, depending on the status of other users. If user *A* is defined as being a friend of user *B*, *A* may know the exact position of *B*.

The scenario also shows that dedicated services must be integrated—in our scenario the LBS service called Squebber. The service represents the interface for applications to interact with. The service provides dedicated querying functionalities and generates SP graphs which are then executed. Therefore, the service itself interacts with other services present in the DSPS, such as an execution environment for SP graph execution. This way a tight integration with the other DSPS components is achieved.

2.4 Resulting Requirements

The sample applications presented beforehand illustrate that many different requirements exist. As an example, we refer to the Rendering operator from Section 2.3.1 which is typically programmed for running on a GPU. An alternative case, in which we do not need dedicated processing capabilities but an adaptation at the infrastructure level, is given in Section 2.3.2. For the scenario of management support in smart factories, it is crucial that sensitive factory data remain within certain predefined boundaries and is not distributed arbitrarily. Another example is the compression of moving object traces from Section 2.3.3, where the integration of specific compression algorithms into the data stream processing pipeline is needed. With the Squebber scenario from Section 2.3.4 it is also important to authenticate users, and adjust the detail level of the data accordingly. All applications are context-aware and adapt their current processing according to their surrounding. They share common principles, i. e. the usage of streamed context data.

In the next subsections we present a classification of requirements extracted from the example scenarios presented earlier. These requirements are subdivided into three classes: *requirements to the DSPS* in Section 2.4.1, *data processing requirements* in Section 2.4.2, and *security requirements* in Section 2.4.3.

2.4.1 Requirements to DSPS

As argued before, applications need specific functionality to tailor the DSPS behavior to their specific needs. We have identified the following requirements:

I-A. **Custom data processing**: Applications often require functionality to tailor the system behavior as well as actual data processing to their specific needs. An example for an application tailoring is the Rendering operator of the visualization application from Section 2.3.1. Thus, custom data processing functionality must be addable to the DSPS. In a large and distributed computing environment such as for DSPSs, it is essential that new operators can seamlessly be integrated into the DSPS, and exploit a tight integration to reduce unnecessary and potentially expensive data transfer routes.

I-B. **Integration of custom services**: Custom services for the interaction with applications or services must be supported (e. g. in the LBS service scenario from Section 2.3.4). This means the data streaming system must support the integration of domain-specific services. Since services might also communicate with other services, adequate mechanisms have to be provided for inter-service interaction. Also, calling other services should be transparent to applications. I. e. if the request is forwarded to another service instance in case the current one is not accepting additional requests, this should happen without further interaction of the application.

I-C. **Dealing with heterogeneous system topology**: Certain tasks may be computationally expensive making the usage of specialized hardware a necessity [107]. Recent research

(such as [134]) shows that creating specialized operators running on Field Programmable Gate Array (FPGA) chips is beneficial. However, the support of such highly domain-specific operators means that DSPSs must cope with a heterogeneous system topology to exploit this potential. This results in a broad variety of participating computing nodes, which must be managed by the DSPS. The operators must also be matched with the available resources. Operators might only be deployed to compute nodes that satisfy their specific requirements.

2.4.2 Requirements to Data Processing

There are requirements that arise at the data processing level. Thus, the DSPS functionality in terms of data processing must be extensible with respect to application requirements. The following data processing requirements have been identified, these applications ask for:

II-A. **Structured and unstructured data support**: New application domains may introduce new kinds of data, such as images or video streams. A mixture of different data structures is of vital importance, as in the application scenario of the location-aware visualization application from Section 2.3.1. On the one hand static data originating from a database and dynamic data from sensors, e. g. modeled as relational data must be supported. On the other hand data formats represented by, e. g. a stream of images must also be supported. Thus, particular care must be taken to avoid conflicts when combining operators at this level. This means that the DSPS must allow to implement new operators that go beyond those provided by state-of-the-art DSPSs.

II-B. **Deployment and execution specifications**: Applications as well as operators may impose certain constraints to the operator deployment and execution. E. g. operators may only be deployed on specific hardware, may require a certain amount of memory at runtime, or may even be allowed to be exclusively executed in a certain (secure) environment, as for the smart factory example from Section 2.3.2. For this reason, the operator model of a data stream processing system must provide a way to describe operators with their respective deployment and runtime constraints. These constraints are defined by operator developers and application developers.

II-C. **Exploiting mobile devices as data source and execution nodes**: In the scenarios described in Section 2.3, mobile devices consume data but also provide data. E. g. for the location-aware visualization pipeline scenario from Section 2.3.1, the client receives a video stream of the rendered scene but must also provide the current position and viewing direction to set the viewport correctly. In contrast, in the trajectory compression scenario from Section 2.3.3 the mobile device provides only data of the current position. Furthermore, mobile devices might even be considered as potential compute nodes when processing data.

2.4.3 Requirements to Security

Finally, requirements concerning the security of a system are important. In the following, two types of entities are distinguished: *Subjects* and *objects*. Subjects refer to entities, such as users of a system or a process running in a system. In contrast, objects are entities for which permissions are defined and usually represent files, database entries, or executable code within a system. Generally speaking, subjects access objects. However, subjects might also be objects. Imagine a subject which wants to change the access conditions of another subject, such as, e. g. a user limiting the sharing of position to only family members (assuming subjects can be grouped according to their family membership). Information can be protected under the consideration of different protection goals, which leads to the so-called *protection targets*. These protection targets influence the actual system design. We have identified the following classification, which is built out of three protection target classes. Each protection target class in turn consists of a variety of targets.

III-A. **Access Control**: Covers all targets that play an essential role for access control. Here a target that is of crucial importance is the data *integrity*, which ensures that objects cannot be changed uncontrollably and therefore guarantee that only subjects allowed to make changes will be able to access this data. Furthermore, the *confidentiality* of information must be ensured in order to hide information from subjects who may not be allowed to read this information. E. g. in the smart factory example from Section 2.3.2, the shift supervisor is allowed to see other data than the assembly-line worker. Where the assembly-line worker is only allowed to see the status of the machine and the current work in progress the shift supervisor also sees work piece related information of the customer who ordered it.

III-B. **Process Control**: This covers all targets that influence the processing of data. It includes the *acceptance* of computation environments which are going to process the data. Since we assume a distributed environment, this protection target defines the computation nodes the data might be processed on. Beside this also data *extent* is of importance. This protection target has influence on the data that is available at a time for data processing by the operators. By limiting the data extent a limited view of the current data window is provided. In this way, a control over the data quality can be achieved. E. g. for the example scenario described in Section 2.3.3, the extent directly influences the precision of the stored traces.

III-C. **Granularity Control**: This protection target class is a special case of the target *confidentiality of information* and *limitation of data extent* in the sense that it describes how fine-grained access and process conditions can be defined. Granularity covers targets which play a role in obfuscating the original data (object) in order to, e. g. prevent conclusions to the subject the data originates from, with techniques such as anonymization and pseudonymization. Other techniques which belong to this protection target class are methods that add some fuzziness to data in order to hide detailed information on e. g. a current position, or aggregate a certain amount of data elements before delivering it to

subsequent operations. As seen from the LBS scenario from Section 2.3.4, friends should be able to know exactly where a user is whereas non-friend users should not. However, if the user wants to navigate to a certain destination, the more accurate the position information is the better the navigation is.

Besides the features mentioned above, also fundamental features such as the authentication of subjects and objects must be supported in order to prevent the abuse of objects. As shown with the LBS scenario, the correct authentication of subjects in the system is essential. This means that each subject must have a unique identity within the system. Thus the identity of subjects must be authenticated to avoid abuse of objects. Authentication covers all targets for the reliable identification of the relevant subjects and objects which are participating in the system. This also includes the *authenticity* of subjects and objects which must have the necessary rights to join the system as well as the action *liability*, which assigns each action to a specific subject. To make these actions traceable a storage area to save the trace information must be provided.

2.5 Foundations

Before starting with the presentation of our solution to the requirements defined above, an overview is given of the underlying technology to understand the rest of the thesis. First, the big picture of Nexus is introduced in Section 2.5.1. Nexus is a management platform to support context-aware applications. In Section 2.5.2, the Nexus federation and its related concepts are presented in more detail, which is the point of origin for the development of NexusDS. Section 2.5.3 describes the correlation of Nexus and NexusDS which together form a context management platform. Section 2.5.4 presents the state-of-the-art in data stream processing and its related work. Section 2.5.5 provides a comparison between Data Stream Processing Systems (DSPSs) and Complex Event Processings (CEPs). Finally, Section 2.5.6 presents the concept of SOA.

2.5.1 Nexus - The Big Picture

Nexus[1] is a project partially founded by the Collaborative Research Center Nexus: Spatial World Models for Mobile Context-Aware Applications (grant Sonderforschungsbereich 627 (SFB 627)). In the context of Nexus [103, 125] many thesis have emerged. E. g. Lange [83] concentrated on the efficient processing of position traces of moving objects for context-aware applications, Nicklas [101] developed the Augmented World Model (AWM) and established its main concepts, or Schwarz [121] developed and implemented the federation of the Nexus platform. Each thesis covers different aspects with respect to the Nexus idea: to build an integrated and open context platform for context-aware applications and to provide mechanisms to integrate and provision context data. This context data is integrated in a federated

[1]The term *Nexus* is a Latin word meaning *connection* or *structure*.

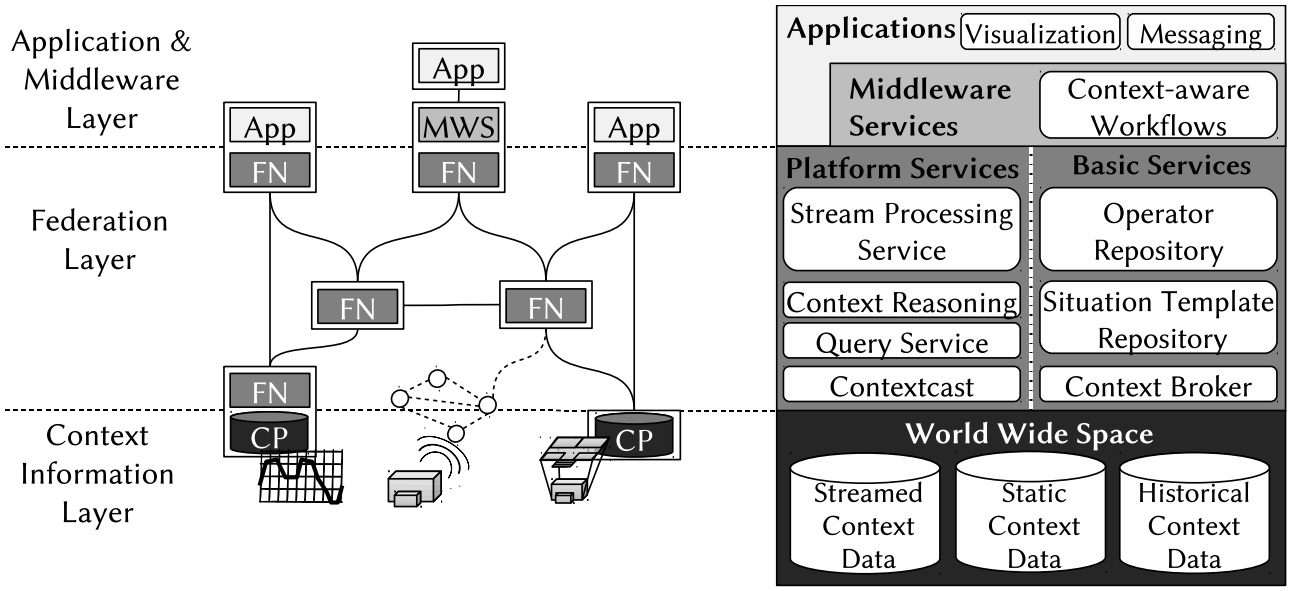


Figure 2.6: Overall Nexus Architecture (adapted from [84]).

context model which is made available through well-defined interfaces. The Nexus platform federates the context models of the different providers to a global context model and offers context-aware applications a global, consistent view on their context data and an integrated data management.

It is highly desirable for such context models to be shared by a wide variety of applications. This desire originates from an economic and a technical motivation. The economical incentive to share such context models is given by the potentially high costs to establish such a context model. The technical incentive is that the context model is consistent and applications can be easily reused with only small adaptations. The vision of Nexus is to provide a so-called World Wide Space (WWS), which is an open and global platform in analogy to the WWW and to enable the seamless integration of context models and processing functionality for a wide range of applications. Therefore, the conceptual and technological framework for such a system has to be researched. The architecture resulting in the Nexus vision is shown in Figure 2.6.

The Nexus vision is represented by a three layer architecture as presented in Figure 2.6, with applications and middleware services that form the top layer. The middle layer constitutes the federation layer which represents a certain set of services for the layer above. Services include a *stream processing service* and a classical *query service* if only static data is requested. Other services included are a *context cast* component to route context messages to appropriate receivers and a *context reasoning* component to deduce higher-level context information by e. g. situation recognition. Between the top and middle layer, a middleware layer is situated. Here *context-aware workflows* offer a workflow service for applications from the top layer, to enable a service-oriented development of context-aware applications. The bottom layer represents the context information layer which constitutes a storage layer for context data. This layer provides access methods to the context data. Depending on the actual context-data

type different storage models are necessary [62]: For *static context data*, such as building outlines, a data base approach seems a reasonable option, for highly volatile *streamed context data* originating from sensors a main-memory storage model is necessary, and for *historical context data* a data warehouse-like storage model is used. To manage all this data Nexus offers an AWM [105]. The AWM is an extensible data model that is based on object-oriented concepts. Both, the Nexus federation which serves as the starting point for this thesis as well as the AWM are presented in the next two subsections.

Such a system yields many challenges. These challenges include: *scalable stream-processing of heterogeneous context data*, a *distributed situation recognition*, *temporal aspects and data histories*, *context-aware workflows*, and *quality of context information*. Each of these challenges constitutes a concrete component in the overall vision and covers a different part of it. To implement the vision of a WWS, as depicted in Figure 2.6, centralized context management systems are obviously insufficient. Therefore, a scalable and distributed architecture is required. This thesis provides the design, implementation, and evaluation of a *flexible DSPS for the domain of context-aware applications*.

2.5.2 Nexus Federation

In this section, we briefly describe the architecture of the Nexus [125] system as it was before this work started. This is also the starting point for the development of the stream processing system—NexusDS—presented in this work, targeting the requirements raised in Section 2.4. In this way, we have extended Nexus in order to be able to process data streams, too.

As depicted in Figure 2.7, the Nexus federation is built up in three layers: an *application layer* containing the actual applications, a *federation layer* containing Nexus nodes, and a *context information layer* consisting of Context Servers (CSs) which provide stored or sensed data. A CS must implement a predefined interface through which it is contacted by Nexus nodes. Furthermore, it must register at the Area Service Register (ASR), announcing the area and object types it offers data for.

The implementation of a CS is not restricted and can be easily tailored to the needs of different kinds of data, like positions of vehicles (high update rates) or the geometry of buildings (large data volumes) [62]. Being an open system, new CSs can be added to the Nexus system. Data of a new CS might overlap with existing ones in both its service area and content, which can lead to multiple represented objects (MReps) [144]. When integrating different result sets from different context servers, Nexus nodes try to detect MReps based on location-based criteria and merge them into a single object [145].

The Nexus platform uses a request-response protocol in which queries are posted in the Augmented World Query Language (AWQL) format, which typically contains a spatial restriction. The result of such a query is a document in the Augmented World Model Language (AWML) representing the result set containing objects that belong to the AWM which is described in

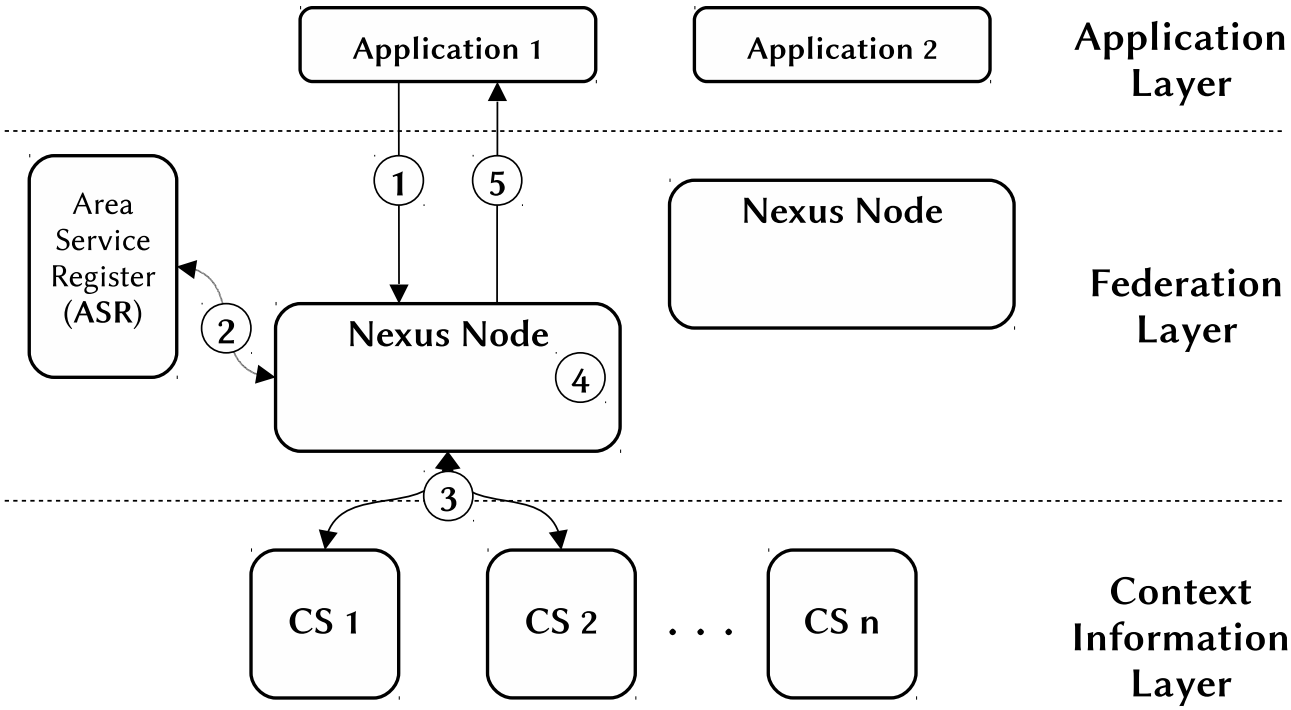


Figure 2.7: Overview of the original Nexus architecture

more detail in Section 2.5.2.1. The processing model is depicted in Figure 2.7 and described as following:

(1) An application sends a query like *"menu and position of all restaurants closer than 1 mile to my current position"* to an arbitrary Nexus node.

(2) The Nexus node determines the relevant CSs by an ASR lookup based on the spatial restriction and the queried object type. In the example above, the spatial restriction corresponds to *closer than 1 mile to my current position* and the object type to *restaurants*.

(3) The Nexus node forwards the query to those CSs. The CSs process the query and send back their results.

(4) The Nexus node integrates the results from the CSs. It detects and merges MReps. For this, domain-specific methods are used that exploit the spatial structure of the data: only objects in a spatial vicinity are considered candidates for being MReps.

(5) The Nexus node returns the integrated result to the application.

2.5.2.1 The Nexus Augmented World Model

To integrate context data from different sources, Nexus provides an extensible data model based on object-oriented concepts: the AWM. The structure of the AWM is depicted in Figure 2.8. The AWM is based on data objects that are formed by attributes. In contrast to object-oriented programming, these data objects do not have methods or behavior. The AWM consists of a Standard Attribute Schema (SAS) and a Standard Class Schema (SCS). The SCS

defines the root set of types that are considered relevant for most context-aware applications, such as *buildings*, *rooms*, or *train stations*. The SAS defines the basic attributes such as *name*, *location*, or *type*. The AWM supports multi-inheritance, objects are instances of one or more object types of the current schema. The schema defines which attributes are mandatory and which attributes are optional for a certain object. An object can have multiple attribute instances of the same type with different values, which, in conjunction with meta data like valid time, allows e. g. the representation of value patterns such as trajectories of moving objects [71]. Supporting multi-attribute instances is extremely important as different data providers might provide various but correct values for the same attribute of an object. E. g. a street might be known as *Theodor-Heuss-Straße* but also as *B27*, and both are correct names for this street. The name, structure and basic data type of the attributes are defined in an attribute schema. A class schema (either the SCS or Extended Class Schema (ECS)) imports an attribute schema and according to the class definitions groups these attributes to object types. The object types in a class schema form an *is-a* hierarchy (inheritance): If an object type *B* inherits from an object type *A*, B has all attributes of A and can define additional ones. Mandatory attributes from A are also mandatory in B, whereas optional attributes can remain optional in B or be defined as mandatory by B.

As depicted in Figure 2.8, context providers or applications can define extended attribute schemas with new attributes and extended class schemas with new object types. These are called Extended Attribute Schemas (EASs) and Extended Class Schemas (ECSs) respectively. ECSs contain sub-types of SCS types, exploiting the object-oriented inheritance concept. As every object of an ECS type can be transformed into an object of a SCS type, those objects are at least partially useful for applications not knowing the ECS. EASs contain extended attributes that are based on basic data types such as string or boolean which are defined in the Standard Type Schema (STS). By this, the components of the Nexus platform can process attributes belonging to standard or extended attribute schemas. New object types defined in the ECSs must inherit directly or transitively from object types from the base class schema. Base schema here might be the SCS or some other ECSs. With this, the Nexus platform can transform objects compliant to the ECS to objects of the base schema by omitting the additional attributes. This allows applications to use an object of any arbitrary extended type by its type in the base schema, losing information, however.

Figure 2.9 is an excerpt for the integration of AWM objects in Nexus, showing how overlapping context-data from two data providers is merged into one single integrated result. Multiple representations are denoted by the same **id** value. Two objects have different type but are multiple representations of the same real world object: **MobileFactoryObject** and **Tool**. They carry different information: One has information about **speed**, the other about the **condition**. The objects also have different values for the **name** attribute. Since the integration layer often cannot decide which value is correct (maybe both) or which one is needed by applications, the merged result contains both attributes, called *multi-attributes*. Both **type** attributes are contained in the merged result, constituting a *multi-typed* object.

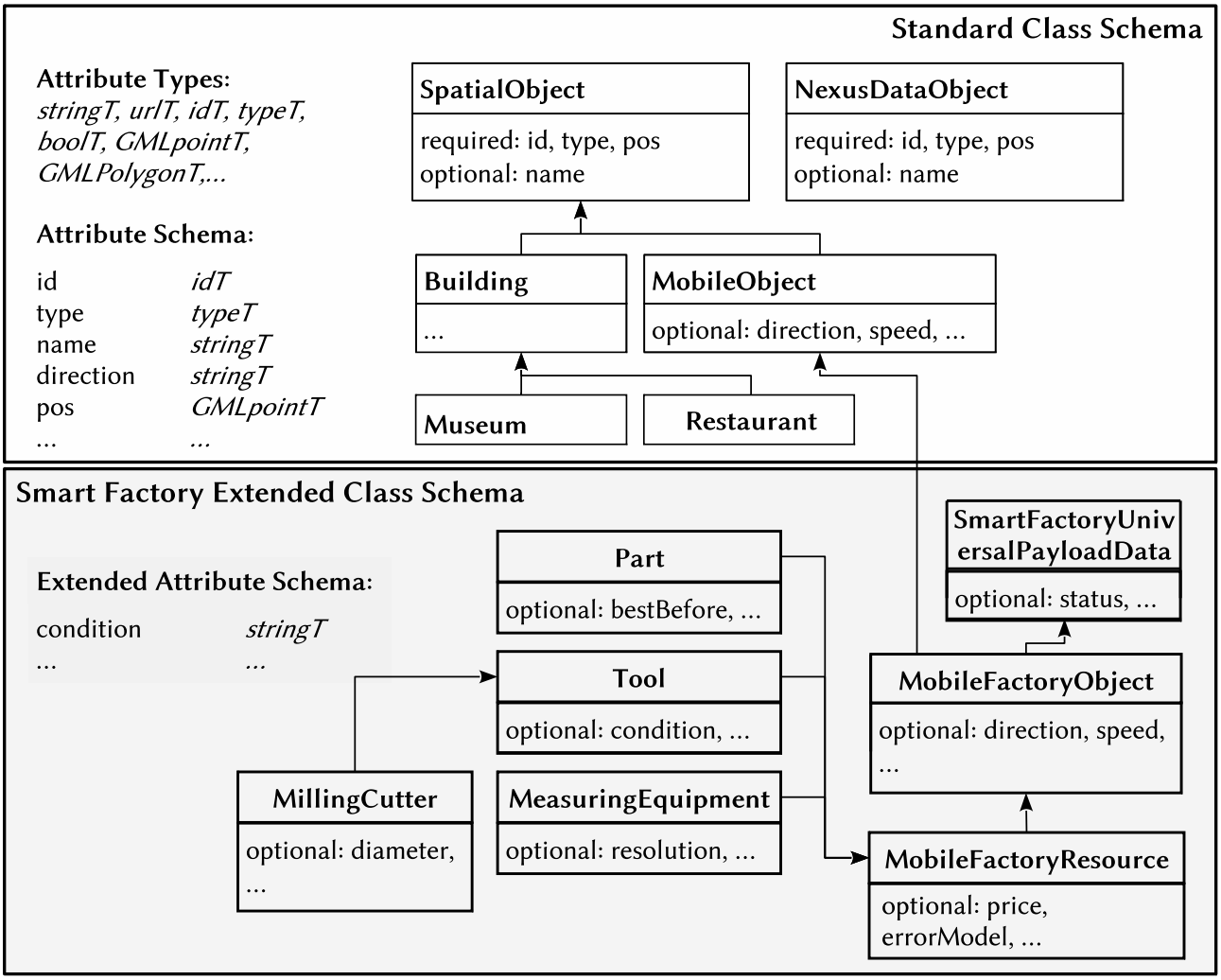


Figure 2.8: The Augmented World Model (AWM). [45]

2.5.2.2 Augmented World Query Language and Augmented World Modeling Language

To formulate queries to the Nexus system, a spatial query language called Augmented World Query Language (AWQL) has been developed, following a query-response-paradigm. To represent the query results, a serialization and modeling language called Augmented World Model Language (AWML) has been developed and used as an interchange format both applications and by platform components.

The AWQL basically supports two DB-related operations: Projection and selection. A projection defines which attributes of the AWM object must be filtered before delivery. A selection provides a way to filter the AWM objects according to a defined predicate. Predicate operators hereby include simple comparison operators such as *equals* or *less than* and highly domain-specific spatial and temporal operators to filter the data. E. g., such spatial or temporal operators restrict the result to those objects lying *within* a certain geographical area or a certain period of time, respectively.

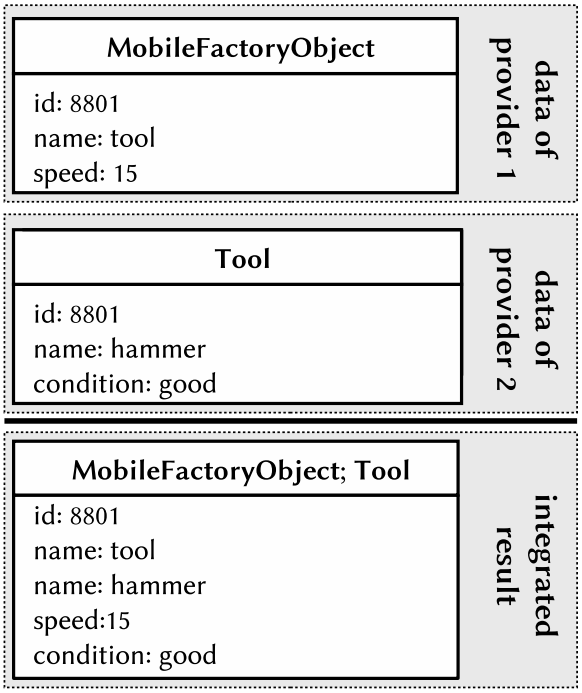


Figure 2.9: Integration example of AWM objects. [45]

The AWML models AWM objects and represents the exchange format for AWM objects. AWML is an XML-based format as shown in Listing 2.1. The excerpt shows a AWML document existing of one root element **awml:awml**, bound to the name space **http://www.nexus.uni-stuttgart.de/2.0/AWML**. Also namespace definitions for the attribute type schema (**nsat**), the attribute schema (**nsas**), and the class schema (**nscs**) are shown. The structure of each object (enclosed by **awml:nexusobject**) is defined by the **nsas:type** attribute. Independently of the actual object type, each object has a unique identifier (ID) called **nsas:nol**. This attribute links to the AWM definition of the corresponding object type and defines its structure in terms of mandatory and optional attributes. The object type in Listing 2.1 is **nscs:Building**.

```
1  <awml:awml xmlns:nsat="http://www.nexus.uni-stuttgart.de/1.0/NSAT"
2          xmlns:nsas="http://www.nexus.uni-stuttgart.de/1.0/NSAS"
3          xmlns:nscs="http://www.nexus.uni-stuttgart.de/1.0/NSCS"
4          xmlns:awml="http://www.nexus.uni-stuttgart.de/2.0/AWML">
5
6    <awml:nexusobject>
7       <nsas:nol>
8          <nsas:value> nexus:<URL>||<AAID>/<OID> </nsas:value>
9       </nsas:nol>
10      <nsas:type>
11         <nsas:value> nscs:Building </nsas:value>
12      </nsas:type>
13      <nsas:kind>
```

```
14          <nsas:value> real </nsas:value>
15        </nsas:kind>
16        <nsas:name>
17          <nsas:value> Museum of Arts </nsas:value>
18        </nsas:name>
19        <nsas:pos>
20          <nsas:value>
21            <gml:Point srsName="http://www.opengis.net/gml/srs/epsg.xml#4326">
22              <gml:coordinates> 48.743668,9.097413 </gml:coordinates>
23            </gml:Point>
24          </nsas:value>
25        </nsas:pos>
26      </awml:nexusobject>
27
28      [ ..... ]
29    </awml:awml>
```

Listing 2.1: Example for an AWML document.

2.5.3 Context Management Platform

Each time we refer to the former Nexus platform for static data processing, we call it *Nexus*. In contrast to this, each time we refer to the Nexus platform for distributed stream processing, we call it *NexusDS*.

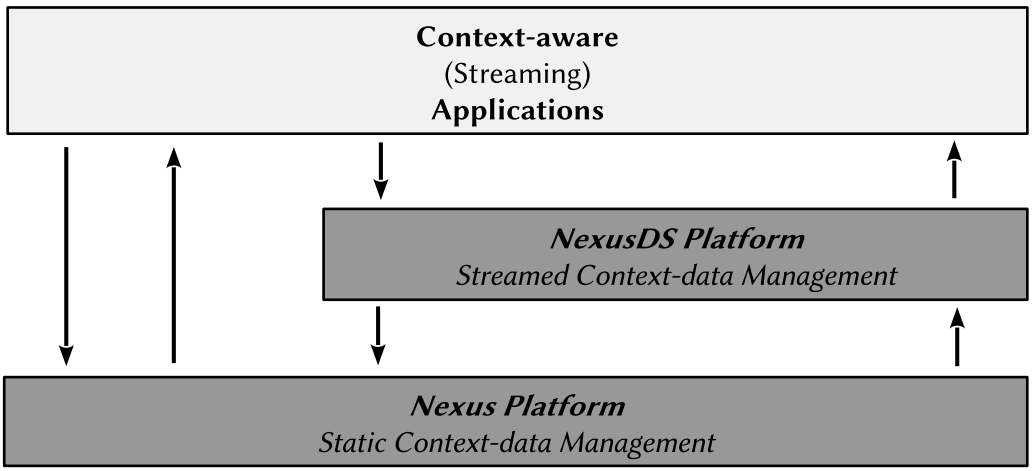


Figure 2.10: Nexus and NexusDS define the *Context Data Management Platform*.

In order to demonstrate how Nexus and NexusDS are related to each other, we have to discuss Figure 2.10. On the upper part, *context-aware (potentially streaming) applications* exist,

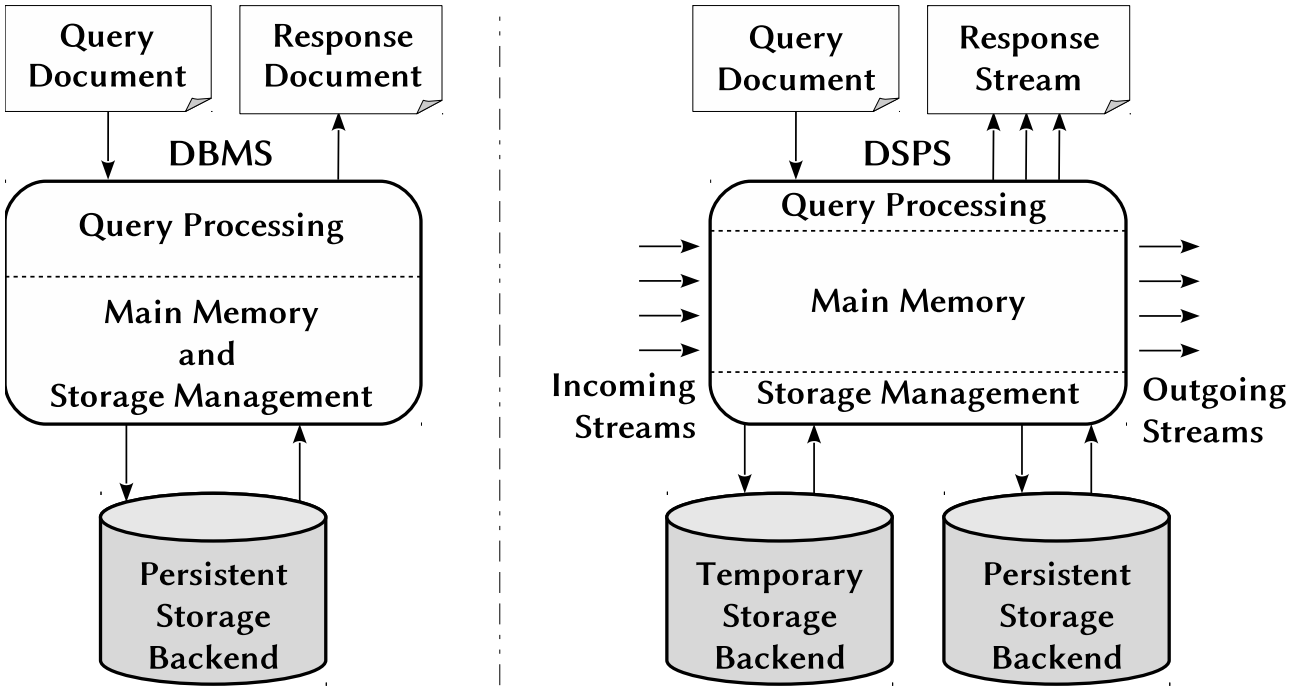


Figure 2.11: Comparison of the architecture of a DBMS and a DSPS.

relying on the functionality of Nexus and NexusDS, which send data processing requests to the respective platforms (either Nexus or NexusDS) and receive the appropriate results. This depends on whether it is an application requesting to process streamed data or static data. NexusDS, which is the DSPS implementing the results of this work, uses the functionalities provided by Nexus to query and receive static data if necessary. This data is then integrated into the streamed data processing in NexusDS. Section 4.7 describes how the data is queried and retrieved by NexusDS.

2.5.4 Data Stream Processing – State of the Art

Data stream processing has been subject to research for more than a decade. This thesis covers the DBMS-oriented perspective on data stream processing. For this reason, most functionalities and ideas of DSPSs are similar to those present in DBMSs. However, many differences exist between a DBMS and DSPS. Figure 2.11 shows these differences as well as dedicated topics related to DSPSs which are described as follows:

On the left side of Figure 2.11 a generic DBMS is depicted whereas on the right side a generic DSPS is shown. For a DBMS data is usually persistently stored in a *persistent storage backend*. The data is assumed to be complete and consistent and a random access to the stored data is possible. Indexes help to enhance and make the access to that data faster. The space available for persistently stored data is theoretically infinite. To access the persistent data usually a *query document* is sent to the DBMS, defining the data of interest (and implicitly also the processing). The query document is computed completely on the current data snapshot before

returning a *response document* containing the exact result. This processing model is called the *pull model* since data is returned when requested, resulting in a single response document. Usually for querying the system, declarative query languages such as SQL are used. However, before accessing persistent data and processing it in the processing area, the data must first be loaded into a main memory buffer area. Thus, only a small part of the data is actually loaded and accessible.

For DSPS a continuous stream of data elements enters the system. A data stream is characterized as a potentially infinite flow of data elements from one or more data sources. This infiniteness makes a storage and postponed processing of the data infeasible since in a worst case scenario we have to wait for infinity to get all data necessary to answer a query. Furthermore, this data usually is time-critical and must thus be processed in realtime. This results in a *push model*, as data is processed as data arrives. This is called a *paradigm shift* from *transient queries* and *persistent data* to *persistent queries* and *transient data*. Data streams are only sequentially accessible and a random access is only possible on small data excerpts (usually in window structures or synopses). These data streams are processed according to a well-defined process definition in form of a query document. Different possibilities exist to formulate query documents. Query documents can be expressed utilizing a declarative query language such as CQL [15]. Alternatives are using a graphical way to define the process definitions [3] or providing a program-like way for defining them [9]. Declarative approaches are easy to understand and to use, even more if SQL knowledge is present. However, they are difficult to extend since the query must be translated to a concrete process definition in terms of an operator tree or operator graph. This can only be reasonably performed with a sound algebra. The program-like approach offers great flexibilities since additional modules can be written to integrate custom functionality. However, a deep understanding and knowledge of programming is needed to succeed, limiting the number of potential users. The graphical approach is intuitive and can be learned easily. On the other side, the efforts to create complex query documents is rather high compared to the other two approaches.

The incoming data streams are processed as they arrive in *main memory*. Sometimes a *temporary storage backend* is needed to store temporary results and unburden main memory. Also, a *persistent storage backend* may be necessary to process a query document. E. g., if we think of a highway scenario where licence plates are scanned, this data must be augmented with additional information that is stored in some persistent storage backend. The processed data streams are then forwarded to subsequent consumers for further processing and a *response document* is continuously produced for the inquirer of the query document, containing the current result. Table 2.1 summarizes the differences between DBMS and DSPS.

A common problem DSPSs have to face is the way streams are treated, since data streams have unlike the traditional way in processing persistent data, uncomfortable characteristics. The biggest problem of all is the potential infiniteness of data streams. Operators which need to process all of its input data before being able to produce output data are called *blocking*. Example operators for the class of blocking operators are *sort* or *join*. Both have a so-called *state*. A state is used to store information regarding data elements processed so far. Stateful

	DBMS	DSPS
Data	persistent data	transient data
Queries	transient queries	persistent queries
Data Structure Operations	arbitrary	append only
Query Result	exact result	may be approximated
Data Access	random access	sequential access
Data Processing	arbitrary processing	one-pass processing
Main Focus	efficient data retrieval	real-time processing

Table 2.1: Feature comparison of a DBMS and a DSPS.

operators are usually of blocking nature. To resolve this blocking behavior the *window* model has been proposed. A window explicitly segments the incoming data stream into discrete and finite snapshots at different timestamps. This allows the application of existing approaches to process the data and solves the blocking behavior of blocking operators. Windows also constrain the used memory by the window boundaries. There exist different window types, such as sliding windows, tumbling windows, damped windows, and landmark windows [3, 110, 156], to name just a few. The size of each window can be either defined by the data element count (e. g. counting 100 data elements), defined by an ordering attribute (e. g. restricting the content by a defined time period), it can depend on a specific predicate (e. g. only objects), or be implicitly defined by *punctuations*. Count-based techniques typically limit the memory needed to store incoming data elements locally, since only a fixed amount of data items is allowed. With time-based and predicate-based windows a memory limit is predictable if the arrival rate of new data elements is considered. Punctuations segment the data streams implicitly, which means it is not done by the DSPS but is rather defined at the producing source. The data source annotates the stream by punctuations which gives the DSPS a hint on how to segment the data stream [141] resulting in a *data dependent window size*. The advantage here is that the segmentation is done at meaningful points. By segmenting the streams in a data dependent manner it is also possible to efficiently implement sorting operations. As mentioned above, sorting operations maintain a state and is consequently blocking. Assume sorting an unsorted stream in descending order by an arbitrary attribute having a natural order. The problem is that we do not exactly know whether we will receive a data element which is greater than the data element with less value. Thus, we cannot decide whether we can forward a partial result of the sorted data stream. Punctuations solve this problem by a data dependent segmentation.

One fundamental problem within DSPSs is the question on how to distribute stream queries across the available resources. In this context the stream query distribution is equivalent to the distribution of the operators a SP graph is actually composed of. Although this topic is closely related to the problem on how to distribute workflow activity calls, it differs. In the workflow scenario so-called Web Services (WSs) are usually bound to a certain host. These WSs offer a service that is callable from outside, e. g. credit rating. In contrast to this, in the DSPS scenario the operators that receive the data (the equivalent in the workflow scenario is the activity) must first be deployed to some computing machine. Thus, the question under concern for DSPS is not how to distribute the calls to workflow activities which process the data transferred data documents, but rather where to place the operators that are going to process the streamed data.

This initial placement—which will be referred to as the *pre-deployment* phase throughout this thesis—typically consists of two phases: A *logical optimization phase* and a *physical optimization phase*.

The logical optimization phase is divided into two steps. First the query is translated into a SP graph consisting of logical operators. In a second step a logical optimization to transform the query into a semantically equivalent one is performed. This optimized SP graph is considered for the next physical optimization phase. During this phase the logical operators are mapped to physical representations of this operator, e. g. a logical join operator is mapped to a physical implementation that is best suited to the current situation. Once done for all logical operators the resulting is a physical SP graph. In a final step this physical SP graph is distributed by placing the physical operators on computing machines and establishing an interconnection between them. Generally speaking, the operator distribution problem is formulated as a Task Assignment Problem (TAP) that is known to be NP-complete. Thus, usually an approximation to solve the problem is used.

The pre-deployment phase is followed by an adaptation phase that is performed during query execution. In the context of this thesis, the adaptation phase is referred to as the *post-deployment* phase. Usually, an exact result in stream processing is not guaranteed due to uncertainties while processing the data streams. Thus, results may be approximated. These uncertainties arise from the network the DSPS is utilizing as a *communication infrastructure* and from software or hardware failures on *computing machines* the computation is performed on. For the former uncertainty type, network congestions might be a cause for bursty data stream delivery rates causing some overload situation on computing nodes. For the latter uncertainty type, a faulty hard disk or operator might be the cause for a compute node failure. Depending on the actual uncertainty type different actions are possible: Reducing the load on a specific compute node by *stream filtering techniques* to reduce the data volume including *precise filtering*, *data merging*, and *load shedding* [34], aggregating data streams in *synopses* [14], and shifting computation on another machine by *operator migration* [155]. Originally, synopses have been developed to implement aggregate functions over data streams. These synopses are also well suited to prevent an overload situation on a machine. In contrast to just dropping data elements, all data elements are considered, resulting in some fuzziness, however.

It is important to note, that we have to cope with compute node failure due to hardware or software problems only operator migrations remain as option.

Different load shedding techniques have been proposed over the last few years. According to literature, the original idea for load shedding goes back to the year 2003 and was first presented in [93] proposing load shedding of random data elements. In the same year, in the context of the Aurora project two different load shedding techniques have been proposed: Insertion of so-called *drop boxes* into the operator network that randomly drop data elements of incoming data streams and insertion of *semantic drop boxes* that drop data elements according to a predicate that consists of QoS functions and system statistics [132]. Later, highly specific load shedding techniques, such as [58, 140], have been proposed solving load shedding for a specific domain of interest. That is what load shedding actually is: It is a highly domain specific technique, mainly depending on the interpretation and the semantics of the streamed data as well as on the infrastructure involved.

If load shedding is not an option (as mentioned above when software or hardware failures occur), the SP graph must be re-scheduled and changes to the current execution must be made. The naïve or straight forward approach would be to re-schedule the entire SP graph and restart with execution. However, this approach is not desirable since already computed results are lost and thus must be recomputed again. Instead of rebuilding all data from scratch only critical parts of the SP graph currently under concern are shifted to alternative computing locations. This process is called *operator migration*. Operators can be distinguished by being either stateful or stateless. Stateless operators are less problematic to migrate than operators with a state. The particularity that matters in both scenarios is that the modified SP graph is semantically equivalent to the original one. This means that no duplicates should be produced and also no results should be missed that would have been computed if no migration had occurred. For operators with state, the state has to be transferred to the new location to avoid false and unnecessary computations. In [16, 60] the problem was first mentioned in the context of DSPSs, but this problem also existed earlier with federated relational DBMSs [100]. For the domain of DSPSs in literature, the first important work in this aspect is [155]. The authors propose two strategies: *Moving states* and *parallel track*. The basic idea of moving states is to move old data elements directly from the old state to the states of the new operators. Sometimes it is necessary to recompute parts of the state for the new operator in order to get a consistent state, matching the old one. In contrast to the moving states approach, parallel track computes the same query on alternative computing nodes in parallel and combines results from both eliminating duplicates. Both strategies are highly implementation-dependent and cannot be arbitrarily applied. Building up on this work, in [151] a generic approach for operator migration has been proposed which combines both approaches, called *HybMig*.

Security in DSPSs is a topic that is currently getting more and more into focus of researchers. This research is driven by the proliferation of social networks, its interconnection with other services such as location-based services, and its pervasion into everyday life as having control over data is getting more and more important. Many persons having a smart phone use social media services to chat with colleagues and friends or share photos with its community. An

answer to the question: *what does the term security in the context of DSPS mean?* is that a DSPS should allow the definition of how data is accessed, who is allowed to access the data, and how data is processed within the DSPS. Access control has long been a hot topic in other research areas, as in the context of relational DBMSs. By the keywords *grant* and *revoke* it is possible for DBMS administrators to grant and revoke access rights to data, according to the Mandatory Access Control (MAC) model [50]. MAC marks all users and data belonging to a certain security level and matches them when access is required. One of the first approaches for the domain of DSPSs is the proposal by [88]. They start from a generic DSPS architecture and extend it by access control modules proposing a generic security architecture for DSPSs. The approach proposes a role based access control. Alternative methods include the rewriting of queries to match access policies [29] and augmenting the stream with security punctuations as proposed by Nehme et al. [99].

2.5.5 Complex Event Processing

A similar approach to the model of a DSPS is represented by the model of Complex Event Processing (CEP). According to Cugola and Margara [48] the models of DSPS [16] and CEP [92] emerged after years of research and are more or less competing nowadays. Cugola and Margara [48] provide a survey of both approaches and propose a classification of so-called *Information Flow Processing (IFP)* systems. The main differences of both approaches are shown in Table 2.2.

Both models have in common, that data can be only accessed sequentially. Thus there is no random access to data. Same as DSPSs, also CEPs have to wait until the data items arrive at the processing unit to perform its operations.

	DSPS	**CEP**
Data	high volumes	high rates
Queries	data transformations	event correlations
Data Structure Operations	generic processing	event notification
Query Result	transformed data	complex events
Data Access	sequential access	sequential access
Data Processing	pipelined transformation	detection rules
Main Focus	real-time processing	event pattern detection

Table 2.2: Feature comparison of a DSPS and CEP.

Also both models share a distributed architecture, which allows to scale out. A distributed architectue is a key feature for efficient and flexible processing of incoming data streams. Same as DSPSs, also CEP distribute operators that cooperate to determine the result of the event correlation process across different machines that perform their dedicated operations. Each operator passes its resulting data items to subsequent operators by a notofication mechanism.

However both system models target different domains. DSPS typically target the processing of high volumes of potentially unbound incoming data streams as DSPSs are data-driven. In contrast to this, CEPs mainly focus the processing of high data rates of incoming data streams as CEP are event-driven. Queries to CEP systems define detection rules that desribe event correlations that are complex events being of interest to the inquirer. Thereby the queries describe event patterns that must be detected by the underlying CEP system. The detected complex events are notified to the inquirers.

In terms of CEP, data items correspond to events that happen in the real world. In contrast to DSPS the processing model of CEPs associate *a precise semantics to the information items being processed* [48]. For CEPs the main focus is the detection of event pattern occurences out of the many low-level events that occurred.

CEPs can be viewed as an evolution to the publish/subscribe model [1]. In publish/subscribe models a subscriber typically subscribes to either content-based or topic-based low-level events. This contrasts to the higher-level and complex events that are deduced by CEP systems.

2.5.6 Service Oriented Architectures and Distributed Systems

The term SOA was first introduced by Schulte and Natis [120] and represents a system model that involves many small computers that are interconnected, in contrast to mainframe computing. The SOA paradigm has many similarities with distributed systems. SOA represents a software architecture for realizing distributed systems, as both constitute a composite network of independent and autonomous entities. Depending on the domain, the term entity might be either a computer (distributed system) or a service (SOA). The SOA paradigm can be defined as being the evolution of the classical client/server architecture from the early 1990s. Formerly, when distributed systems emerged and the client and server paradigm was the main method of communicating, clients inquired a certain service which the server offered. Therefore, the client had to know about the server to get the service of interest. They also needed some common "language" in order to communicate. With SOA, these small computers offering services interact with each other by message exchange, typically formulated in some XML-based dialect.

The SOA system model gives great flexibility when creating systems or applications. A server offering some service could also be a client that utilizes a service present on some other server. This is similar to the Peer 2 Peer (P2P) [119] idea but SOA focuses on higher level interactions, service composition, and it represents an abstract architecture paradigm in contrast to low-level interaction protocols as described by P2P systems. In the SOA world, systems

(or applications building up on the SOA idea) are not hard-wired. The SOA paradigm allows the flexible adaptation to new requirements and changing conditions by integrating additional services. This SOA philosophy is applied in many domains. A prominent example making massive use of the SOA idea is represented by the domain of Enterprise Application Integration (EAI) [90], which has evolved to service integration in Enterprise Service Bus (ESB) architectures [33]. Generally speaking, services encapsulate well-defined functionalities in independent, self-describing, and autonomous modules. Services might also be arranged to create composite services to realize higher-level functionalities. There is no clear definition of major concepts a SOA must fulfill. The term SOA is used in different scenarios, each possibly having different requirements to the actual implementation of the SOA with respect to a specific problem. From our perspective, the major principles of a SOA are as follows:

- **Services are loosely coupled**: Services usually are loosely coupled which means they typically interact with different services in their environment. These services are instantiated on arbitrary computers within the network. Thereby, we differentiate between service classes and service instances. A service might depend on a certain service class, but does usually not on a specific service instance.
- **Services are self-describing**: In order to integrate services into an existing system infrastructure the service must have the ability to describe itself. We refer to this self-description as *service meta data*. This meta data is exploited by the system to find suitable services.
- **Services are reusable**: Services are not bound to a certain computer. A service can be instantiated and offered on different computer nodes. The only exception to this is provided by the service meta data mentioned beforehand. The meta data can restrict the execution of a certain service to a certain computer, e. g. for security reasons.
- **Services are discoverable and location-transparent**: As mentioned above, services are loosely coupled and might depend on other service classes rather than on service instances. This in turn means, service classes as well as service instances must be discoverable. Furthermore, the actual location a certain service instance is running on must be transparent to the inquirer. The discovery of service classes reveals which services are available. The discovery of service instances yields where a service instance of a certain service class is currently running thus accepting requests.
- **Services are autonomous**: A service encapsulates functionalities which are fully implemented by this service. No additional modules are necessary. However, services might rely on specialized hardware which must be present on the system executing this service. This is a stringent requirement of a service and is described by the service meta data.

Although these five principles described above are sufficient for our domain, it is important to note that the selection of principles does not claim to be universal w.r.t. the term SOA. Depending on the domain of interest there might exist even more principles that can be applied to the term SOA, such as *services can be composed out of other services* or *services are self-describing*.

2.6 Summary

In this chapter, applications are presented and their requirements are discussed. These applications, although originating from different application domains and having different processing constraints and security concerns, share a common processing principle, i. e. data stream processing. They are often embedded in an utterly heterogeneous and distributed infrastructure ranging from desktop computers to mobile devices. Therefore, a data stream processing system needs to supply adequate techniques to provide a tight integration of standard and non-standard processing schemes. These techniques reach from the integration of specific operators, through the definition and manipulation of the actual SP graph deployment, to the integration of domain-specific service functionality. Concepts and mechanisms have been designed and created to build a flexible DSPS, which is able to support a broad variety of context-aware applications. We have integrated the concepts and developed a DSPS called NexusDS which is presented in the next chapter, and is especially tailored to meet these requirements. By the unique techniques provided by NexusDS it is possible to realize applications relying on highly domain-specific concepts.

Part II

System Architecture and Data Processing

3 System Architecture

After the discussion of the data stream application scenarios and the analysis of their requirements towards a DSPS in Section 2.3, in this chapter we present the DSPS architecture and its components.

In Section 3.1, a short introduction is given summing up important facts from previous sections. Afterwards, in Section 3.2, the necessity for adaptation in DSPSs is discussed. Related work is presented and discussed in Section 3.3. Section 3.4 presents *NexusDS*, our DSPS to target the requirements detected. First the compliance of NexusDS to the requirements raised is discussed and the basic architecture and its main components are detailed. This chapter is concluded by a short summary in Section 3.7.

3.1 Introduction

Data stream processing has been in the focus of many researchers and is still a topic of high interest. Research stretches from system architecture proposals over adjusted stream processing techniques to query distribution and reuse, resulting in more and more sophisticated techniques. As a data stream is characterized as a potentially infinite flow of data elements from one or more data sources, data stream processing is typically done in two steps: Data elements are collected from the data sources and are processed according to some processing definition consisting of a clearly defined set and order of operators. Complex scenarios that go beyond the current state-of-the-art have been presented in Section 2.3.

To satisfy the needs of specific application domains, such as visualization, a DSPSs must be extensible towards, e. g. specialized operators which may require specialized hardware. Each application, or more generally application domain has one or more requirements according to Section 2.4. Thus, an *adaptation requirement* of DSPSs exists, which means that the DSPS must provide concepts to adapt to these specific requirements: (I-A) *custom data processing*

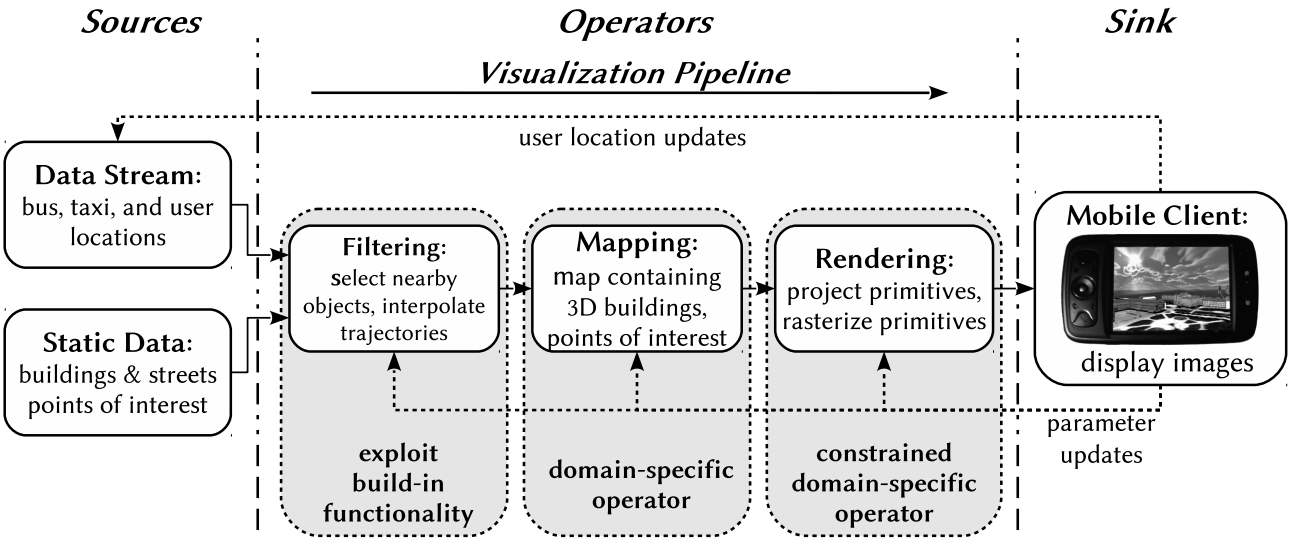


Figure 3.1: Simplified Operator Graph of an Interactive and Location-Aware Visualization Pipeline for Mobile Client Devices.

logic, (I-B) *integration of custom services*, (I-C) *dealing with heterogeneous system topology*, (II-A) *structured and unstructured data support*, (II-B) *deployment and execution specifications*, (II-C) *exploiting mobile devices as data source and execution nodes* as well as (III-A) *access control*, (III-B) *process control*, and (III-C) *granularity control* for data.

To address these new requirements we have developed a highly flexible and distributed DSPS targeting these requirements. This DSPS is referred to as *NexusDS* throughout this thesis. It should be noted that although the DSPS architecture and its related concepts is referred to as NexusDS, the concepts presented are general. The concepts presented can be used in order to create a flexible DSPS that targets the requirements from Section 2.4, which are imposed on a flexible DSPS.

NexusDS features a *flexible operator model*, *constraint-based operator distribution and deployment*, *management of heterogeneous system topologies*, *data access control mechanisms*, and a *SOA-based P2P approach for dynamic, on-demand scale out.* The goal of NexusDS is to support a large variety of context-aware applications as presented in Section 2.3. Other example scenarios are conceivable however, like location based information systems [102], mobile games [104], or even factory management applications [47]. Thereby, NexusDS is based on the AWM, a shared, global context model as presented in Section 2.5.2.1.

3.2 Adaptation Requirement

Nowadays distributed DSPSs are state-of-the-art since they scale well with increasing workload and thus allow an efficient processing. These systems provide an interface to the user who defines how data is processed by the system. Incoming data streams are processed continuously in a distributed fashion across available computing nodes.

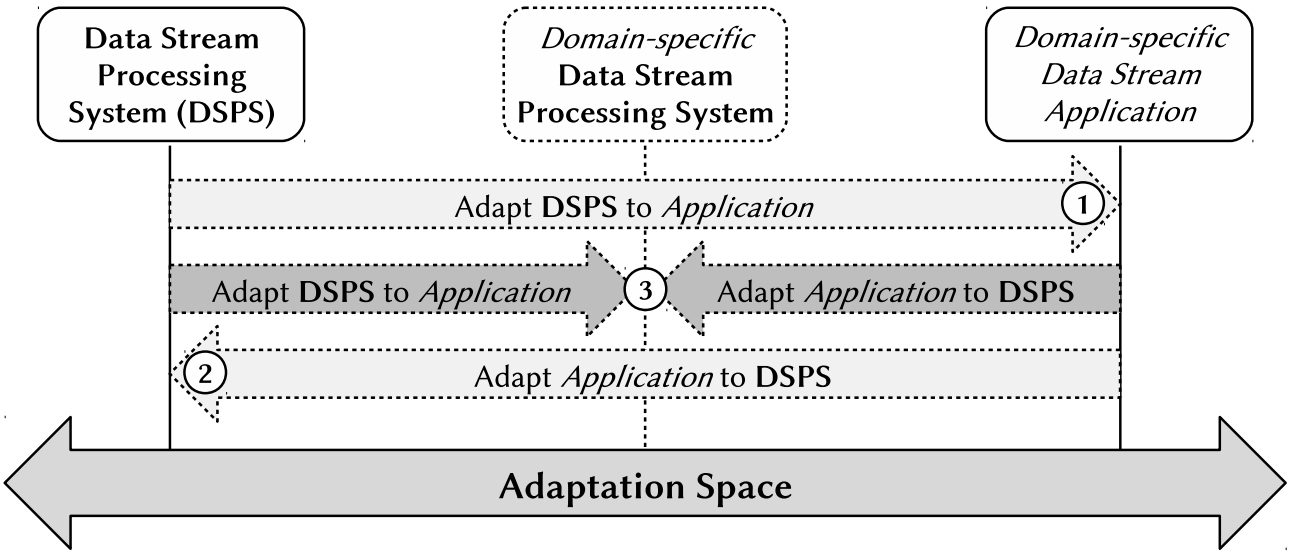


Figure 3.2: Adaptation Problem for Domain-Specific Data Stream Applications

Figure 3.1 shows the example scenario of a location-aware visualization pipeline. The final result of the process is an image stream showing the surrounding of a possibly low-performance mobile client device. First, the streams of the continuously changing user, bus, and taxi positions and the static geographic objects are merged and filtered on the restriction of the user's proximity. Then, the mapping step assigns graphical primitives to the filtered data, such as triangles, points, etc. Finally, a rendering operator transforms the stream of graphic primitives into an image stream that can be displayed on the client. Preferably, the rendering step is executed on specialized graphics hardware (GPU). User interaction, such as rotating and panning the scene, can be modeled as parameter updates for the operators, e. g. the camera parameters of the rendering operator.

To deal with the distributed nature of the different data streams in this scenario, it is clearly desirable to make use of a DSPS. A *visualization pipeline*, as depicted in Figure 3.1, generally consists of three steps: *Filtering*, *Mapping*, and *Rendering* [65, 154]. These steps map nicely to complex operators of a stream processing system. However, this is far from reality since DSPSs are not designed to support domain-specific applications. The operator's origins are different and so are their respective constraints. *Filtering* can be realized by *exploiting already existing operators* (from a DB's point of view a *selection*) present in state-of-the-art DSPSs. For *Mapping* domain-specific knowledge is necessary. Mapping does not need a specific execution environment to run, in contrast to *Rendering*. As with Mapping, also Rendering is a domain-specific operator but has constraints with respect to the target runtime environment making a *constrained domain-specific operator* out of it, i. e. a GPU is needed.

According to the sample scenario the DSPS must adapt to new requirements in terms of operators, changing deployment and runtime specifications, or even changing system topologies and security restrictions. Although research has brought up many efficient techniques,

an *adaptation problem* in the domain of DSPS is still persisting. The adaptation problem is depicted in Figure 3.2 and is explained ahead.

On the left side a generic DSPS is depicted (in bold letters), whereas on the right side a domain-specific and context-aware data stream application is displayed (in italic letters). In principle three approaches exist to integrate such an application into DSPSs to exploit existing functionality: Either ① *adapt the DSPS* to meet the specific requirements of the domain-aware application, ② *adapt the domain-specific application* to exploit system functionality where possible, or ③ *adapt both*, the DSPS and the context-aware application, meeting somewhere in the middle. This case is depicted by the *Domain-specific DSPS*.

Case ① considers adapting a DSPS to an application. In this case, a DSPS must either be built from scratch or an existing one must be modified to achieve a deep integration of the application in question into the DSPS. Neither of these two alternatives is desirable, as building a DSPS from scratch means a big investment—in terms of time and maybe money—and includes a high failure probability. To extend an existing DSPS or even building up a DSPS from scratch needs a profound knowledge of the system specifics. A prominent example is PIPES [79], a framework offering building blocks that must be implemented with domain-specific code. But before the system can be used, its missing features must be implemented by an expert. This, however, is a non-trivial and cumbersome task.

For case ② the domain-specific application must be adapted to an existing and concrete DSPS. This reduces the re-usability of the domain-specific application automatically since the application is tailored to the specific DSPS. This is because the application might rely on specific functionalities which are only offered by a certain DSPS. In this scenario, DSPSs capabilities are exploited to carry out tasks a DSPS can handle. The domain-specific data processing must then be performed within the application or outsourced to other, maybe remote, sites. However, an obvious issue with this approach is potentially redundant code which is spread across many different applications, in each specific domain. Another problem is the fact that devices running domain-specific applications are often not well-suited to do so. This mainly originates from computing performance and energy constraints.

Considering case ③, the domain-specific applications as well as the DSPS are adapted to meet *somewhere in the middle* (denoted by the dashed box in Figure 3.2). Therefore, the application outsources parts of the domain-specific application logic to integrate it into the DSPS. In turn, the DSPS must consider the domain-specific constraints of the application. Assuming the user wants the application to be executed exclusively on nodes he trusts for security reasons, the design of a DSPS is crucial for the flexibility with integration of domain-specific applications. This has mainly two reasons: The integration should be as easy as possible unburdening the developer from cumbersome tasks. At the same time, the system must provide adequate extension mechanisms in terms of extension points. NexusDS is our proposal of such a DSPS which provides basic data stream processing functionality, but at the same time provides extensions for the integration of domain-specific functionality.

Each domain-specific application has different requirements towards the system. E. g. they need a GPU to perform certain application functionality or require data that should not cross a certain administrative domain. Even users using a domain-specific application may have different preferences, e. g. limiting the execution to a set of execution nodes they trust. These *constraints* imply a possibly different execution of SP graphs as DSPSs would usually do, i. e. the system must cope with these constraints occurring in such heterogeneous and distributed systems.

3.3 Related Work and System Classification

Before going into details of the NexusDS, related work in the domain of DSPSs is presented and classified in this section. Complex data stream application scenarios need adequate mechanisms that go beyond the ones present in state-of-the-art DSPSs, as discussed in Section 2.3 and their resulting requirements to modern DSPSs from Section 2.4. In this section, up to this date some of the most relevant DSPSs are classified. They are compared to each other and their specific differences are highlighted.

Aurora [3] has been a common effort of the universities Brown and Brandeis and the MIT and goes back to the year 2003. Its main characteristic is the integration of *user-defined functions (UDFs)*, which allows to add custom data processing capabilities to the system. However, due to its monolithic—thus *centralized*—system topology, only a limited amount of queries are processable in parallel. The *relational model* is used as basic data structure and queries are formulated by a *boxes and arrows* concept where a box is a stream operation (either built-in or a UDF) that is interconnected by arrows with other boxes forming a stream operation network. This operation network, however, is only executed locally since no distribution is possible. In the year 2009 Cao et al. [29] extended Aurora by a data access control mechanism called *ACStream*. By ACStream it is possible to control data access. Thereby, access control restrictions are defined as *predicate expressions* that describe an explicit assignment of access rights to certain subjects. To illustrate this, imagine a data stream representing positions where each data element has an **ID** attribute and a **location** attribute. An expression in ACStream can define read access for a subject **A**, if the ID attribute has a certain value or the position is within a defined quadrant. A special feature of the concept is the possibility to define temporal constraints by *time-based windows*. Access to data elements can be restricted to a certain time interval via temporal restrictions. The start time and end time can be explicitly defined and the time interval is provided by defining the absolute time interval size and the interval step size. ACStream enforces access control by rewriting queries. The rewrite process possibly selects *security operators* instead of regular operators to carry out the defined access control constraints. Four different types of security operators are available: *Secure View* processes an input stream by applying the access restrictions and returning a view of the data stream with only data elements meeting the access restrictions. *Secure Read* operators filter data elements and remove attributes, *Secure Join* operators filter the output data streams composed

of multiple input data streams, and *Secure Aggregate* operators control aggregate functions. A fine-grained way for defining the granularity in which data can be accessed and processed is not supported, resulting in an *all or nothing* semantic.

In the year 2003 researchers of the Stanford University started the development of *STREAM* [14]. STREAM stands for ***St**anford St**re**am Dat**a** **M**anager* and is—as Aurora—a *centralized* DSPS. In contrast to Aurora, STREAM does not allow to integrate any custom data processing functionality. As with Aurora, STREAM also utilizes a *relational data model*. The main characteristic of the STREAM system is the capability to integrate static data into the stream processing as well. The fundamental idea is to treat all data as relations and to apply relational processing techniques to process them. However, as mentioned in previous sections, a stream is potentially unbound. Therefore, special operators have been developed that transform either a stream to a relation (*Stream2Relation* operator) or a relation to a stream (*Relation2Stream* operator). Doing so, the well known relational processing techniques can be exploited and paired with the emerging domain of stream processing while avoiding the problems related to data streams. Beside the other limitations of the STREAM system, the main limitation is the centralized system structure not allowing to scale out processing power as needed. STREAM does not provide any security relevant mechanisms to prevent misuse of data.

In 2005 Kuntschke et al. [80] proposed *StreamGlobe*. It was one of the first systems capable of processing data streams in a *distributed* fashion. StreamGlobe provides a fixed set of operators at each computing node depending on the actual computing node type. StreamGlobe categorizes computing nodes into Thin-Peers, Thick-Peers, and Speaker-Peers. Thin-Peers perform simple processing tasks. Thick-Peers may perform advanced processing techniques, whereas Speaker-Peers may additionally optimize the queries. Custom operators must be *included with each query* that is submitted to the system and are only valid for the particular query. Subsequent queries have to send the custom operator again in order to use it limiting the usability of custom operators to the respective query only. StreamGlobe utilizes the *relational model* as data structure. StreamGlobe is able to integrate mobile objects as *data sources* into the data processing tasks.

In 2007, Xiong et al. [150] proposed PLACE*. PLACE* is a spatio-temporal *distributed* stream processing system for moving objects. It is based upon PLACE [96] which is a scalable location-aware database server. Unstructured data is not supported, the data model used is the *relational data model*. PLACE* does not consider the integration of custom operators but offers spatio-temporal operators, such as *INSIDE* and *k-nearest neighbors (kNN)* operators. The authors have also introduced so-called *negative tuples* that allow an incremental evaluation of queries. By this technique, data elements which have already been processed and delivered to the inquirer can be removed in retrospect. The whole query execution is performed in backbone servers that also track the moving objects as they move along. This means mobile devices are exploited as *data source* but not integrated with the actual query execution. A weak point, however, is the missing support for access restrictions to position data of moving objects.

In 2005, one of the most influential works called *Borealis* has been proposed by Abadi et al. [2] as a progression of Aurora [3], Aurora* [38] (a distributed version of Aurora), and Medusa [152]. In contrast to its predecessors, Borealis has its main focus in processing data streams in a *distributed* fashion which was the major limitation of Aurora. Aurora* provides first building blocks and experience for processing data streams in a distributed fashion but was not developed further at that time. Borealis provides a programming interface to implement and integrate custom data operators into the system. The operators have to be placed on a specific desktop computer by an administrator and are only available on this computing machine, which means the operators are only *locally available*. As Aurora does, Borealis also supports only structured data and builds upon a *relational data model* to represent data. In 2006, Lindner and Meier [89] extended the Borealis engine to permit access control of data by including additional components. To achieve secure access, a session management component and an authentication component have been integrated into Borealis without modifying the original system. If access to a certain data element is not granted, it is eliminated at the end of the processing and before delivering the data element. Data access can be specified for each subject. However, it is only possible to make the decision whether or not the subject is allowed to access the data element with an *all or nothing* semantic. A fine-grained level-of-detail setting is not provided. Secure Borealis supports *encryption* between processing nodes. To enforce the security policies, the data is filtered—in contrast to the actual processing which is fully distributed—by a *centralized* component. This circumstance is a potential bottleneck and represents the possibly single point of failure since all data has to pass through this component before it can be forwarded to subsequent operations or the target. This in turn means that the access control enforcement is performed after the final data elements are determined, i. e. after the entire query processing is done. This strategy might discard costly calculated data resulting in a waste of resources. A commercial version that covers the functionality proposed by Aurora and Borealis is also available and is called *StreamBase* [128].

PIPES [79] is a framework which was developed at the Universität Marbug in 2004, and its main focus is to provide a framework that allows implementing a custom DSPS. PIPES is composed out of basic modules that can be used and extended to build custom DSPSs. These modules already implement core functionalities, but must be plugged together by a system developer depending on a certain area of interest, to build a working DSPS. This means that PIPES allows to flexibly develop custom DSPSs by providing certain functional blocks at the same time. This, however, requires the system developer to have a profound knowledge of the special data stream processing principles and properties. PIPES offers basic operators based on a *relational model*. System developers can implement additional operators. All operators must be installed at a specific remote site running the system and as a consequence the operators are only *locally available* at that specific computing node. As a querying mechanism PIPES supports CQL and provides a compiler to translate the CQL query to an equivalent operator tree in PIPES. A commercial version of this research prototype is *RTM*, which stands for ***R**eal **T**ime **M**onitoring* [115].

Sutherland et al. [129] proposed *D-CAPE*, which is a further development of *CAPE* [116]. CAPE is a centralized DSPS which utilizes a *relational data model* to represent its data. Its main characteristic is the combination of windows and punctuations. Usually, up to this point in time windows partitioned the data stream according to a defined extension, i. e. time or object count. By exploiting punctuations the window is constrained by the data stream itself. This idea has been retained for D-CAPE which constitutes a *distributed* version of CAPE. D-CAPE relies on a shared-nothing paradigm which guarantees more efficiency with pipelined parallelism [129]. The main difference between CAPE and D-CAPE is a distribution component which manages the distribution of the query. D-CAPE utilizes a defined set of operators to process data streams of a specific format and does not provide any operational extension mechanisms. System extensions in terms of additional services are not planned. However, developers have the opportunity to integrate custom knowledge in form of a *distribution pattern* to influence runtime adaptation tasks. After an initial operator distribution—which is not adaptable—a distribution pattern provides the way the adaptation of the running queries should occur, basically being a distribution rule to adapt query execution. As those, D-CAPE provides a so-called *round-robin distribution* and a *grouping distribution*. The system mainly focuses on desktop computers with no specialized capabilities. D-CAPE does not provide data access control mechanisms or other security-related functionalities.

Odysseus [13, 26] is a modular DSPS framework. Its main idea is to provide a solid flamework that already works out of the box but also allows custom extensions. These extensions can be plugged into Oysseus via so-called *variation points*. These variation points are available for physical operators, query translation, restructuring, transformation and execution. Custom operators can be added to Odysseus and along with the operators extensions to the translation, restructuring and transformation modules. The translation module allows to extend its functionality by additional mapping rules that map a declarative query to a logical operator graph. The restructuring module accepts a set of restructuring rules to create a semantically equivalent logical operator graph. The transformation module maps the logical operators to physical ones. The mapping of the restructured logical operator graph is based on a cost model [26] and the physical operators are interconnected via a publish/subscribe mechanism. To the best of our knowlege, Odysseus provides no possibility to influence the placement of the physical operators. But it is possible to influence the operator scheduling during runtime via the variation point of the execution module. To do so, buffer elements are placed between two physical and interconnected operators. This approach is similar to PIPES [79]. The scheduling is based on a predefined Service Level Agreements (SLAs) policy [143]. Thereby the placement decision of the buffer elements greatly influence the system's overall performance [26]. Odysseus provides a monotoring module that allows to visualize the current system's state. Odysseus does not provide data access control mechanisms or other security-related functionalities.

The last DSPS considered for classification is *SystemS* [9], which was developed by IBM Research. SystemS has also a commercially available variant known as *Infosphere Streams* [21, 72]. For this thesis the research prototype called SystemS is considered as it covers the main aspects that are also available in the commercial version of the system. The

main focus of SystemS is on data mining. The main idea is to apply data mining algorithms to data streams from multiple sources (mainly sensor sources) and derive higher level information and filter unnecessary data. Thus, it mainly focuses on data mining applications. But SystemS is also extensible as it allows to integrate custom operators into the system. These custom operators must be installed by a system administrator and are only *locally available* at the computing node they have actually been installed on. SystemS has a *distributed* system topology, and machines running operators range from ordinary desktop computers to mainframes. Beside structured data in form of *relational data* also arbitrary *unstructured data* is supported by SystemS. To formulate a concrete SP graph, operators are arranged into an operator network. Thereby operators are not directly interconnected but are rather dynamically interconnected by a publish / subscribe mechanism. The source operator publishes its information on what data it delivers and the consuming operator subscribes to that data descriptions. Application developers can deploy SP graphs in SystemS in two different ways: By either leaving the decision to the operator scheduler of SystemS or by *defining the operator deployment completely*. This means that the application developer cannot define a partial deployment as NexusDS allows, e. g. to fix certain SP graph fragments. SystemS also supports secure connections between the computation nodes and allows to regulate access control to data by *encryption*. In order to constrain interaction of the operators with the "outside world", e. g. network access, specialized operating systems (OSs) are necessary such as *SELinux*[1]. This means that enforcement of operator constraints can only be achieved by a deployment to computation nodes running the specialized OS. It is also possible to restrict the execution of operators and thus the processing of data to a number of security domains (running the specialized OSs). This process is called *Processing Element containment* (*PE-containment*). A fine-grained access to the data is not provided by SystemS resulting in an *all or nothing* semantic.

To conclude the classification of the related work, *NexusDS* is the system developed to consider the requirements raised in Section 2.4. NexusDS shares a few concepts with the systems discussed, but differs in several ways. NexusDS also allows the extension of an existing operator base with additional custom and highly domain-specific ones. The operator model in NexusDS differs from existing approaches as operators published in our system are available at a global scope and are not limited to a certain computing node where they must be first installed. Therefore, operators are published via an operator repository holding the operators and providing a querying mechanism to find out which operators are available. This gives a great flexibility to deploy operators on computing machines when needed. Besides the operational extension, also functional extensions can be made to the system by providing custom services that can be loaded by NexusDS and made available within the network. This is beneficial if we imagine a domain which utilizes a dedicated query language and needs support for mapping these specifically formulated queries to those supported by the NexusDS. NexusDS supports a large variety of devices ranging from simple mobile devices with a reduced set of capabilities to desktop computers with dedicated hardware installed. This differentiation is of great importance when looking for suitable devices capable of executing an operator since NexusDS makes

[1] **http://www.nsa.gov/research/selinux/**

	(I-A) Cust. Operators	(I-B) Ext. of System Func.	(I-C) System Topology	(II-A) Struct. / Unstruct. Data	(II-B) Deploym. and Exec.	(II-C) Mobile Devices	(III-A) Access Control	(III-B) Process Control	(III-C) Granularity Control
Aurora	user def. functions	-	centralized	relational / -	-	-	predicate	time-bas. window	all or nothing
STREAM	-	-	centralized	relational / -	-	-	-	-	-
Stream Globe	provided with query	-	distributed	relational / -	-	data source	-	-	-
PLACE*	-	-	distributed	relational / -	-	data source	-	-	-
Borealis	local operators	-	distributed	relational / -	-	-	encrypt. / central.	-	all or nothing
PIPES	local operators	-	distributed	relational / -	-	-	-	-	-
D-CAPE	-	-	distributed	relational / -	distribution pattern	-	-	-	-
Odysseus	global operators	algebra extensions	distributed	objects / arbitrary	-	data source	-	-	-
SystemS	local operators	-	distributed	relational / arbitrary	complete definition	-	encrypt.	PE-containment	all or nothing
NexusDS	global operators	global services	distributed	objects / arbitrary	partial definition	data source & execution	encrypt. / SI-filter	nodes / window	LoD filters

Table 3.1: Comparison of data stream processing systems.

no assumption on the actual operator implementation, perhaps relying on highly specialized hardware which must be available in order to bring the operator to execution. Thus, we also need to characterize the single operators w.r.t. their hardware and software requirements as well as other specific constraints such as restricting the execution to a certain subset of nodes. This makes the definition of deployment and execution specifications a necessity. Furthermore, NexusDS allows to process structured data streams in form of AWM data objects as well as arbitrary unstructured data streams. This basic object structure is retained as a fundamental data structure since we aim to extend the already existing Nexus platform by data streaming capabilities. NexusDS provides facilities to enforce data access control and data process control in order to avoid misuse. Therefore, NexusDS implements an authentication mechanism to distinguish different participants within the system and to coordinate data access. Besides data access, also process control policies can be defined stating how data originating from a certain source can be processed and by whom. Correlated to this, NexusDS also supports granularity control policies for a fine-grained data access and data process definition. By defining so-called LOD-filters, the original data is transformed in a way to make data processing possible, even if the user requesting the data is not allowed to access the whole data records, but only parts of it. In this case the banned areas are eliminated and the user is only provided with data he is actually allowed to have access to.

All systems considered in this section allow access of data, and process them mostly in a distributed fashion which is known to be a key feature to efficiently process data streams. Two centralized systems are considered, namely Aurora and STREAM, as they have been one of the first proposals and also the most significant centralized approaches in this research area. Table 3.1 sums up the related work discussion and compares the state-of-the-art in DSPSs according to the requirements formulated in Section 2.4.

3.4 NexusDS – Flexible Data Stream Processing

As discussed in Section 3.2, an adaptation requirement exists which conditions an open platform to integrate various data sources as well as processing techniques by exploiting already existing functionality at the same time. NexusDS builds on the Nexus system [103] which is described in Section 2.5.2. Nexus is an open platform for context-aware applications originally designed for the query-response paradigm. With the proliferation of streamed data, such as for mobile devices with all their sensors providing data streams or the mining of monitored streamed data to enable intelligent transportation systems [25] in real time, a necessity to support their efficient processing becomes mandatory and the importance of DSPSs in this field grows. However, DSPSs should be capable of adapting to changing system conditions as demanded by the application domains. NexusDS constitutes an open DSPS and is able to adapt to special needs and at the same time provide all necessary parts to start over, without the necessity to implement components in order to make the system actually work. NexusDS uses the powerful AWM (see Section 2.5.2.1) of the Nexus system without modifications. NexusDS

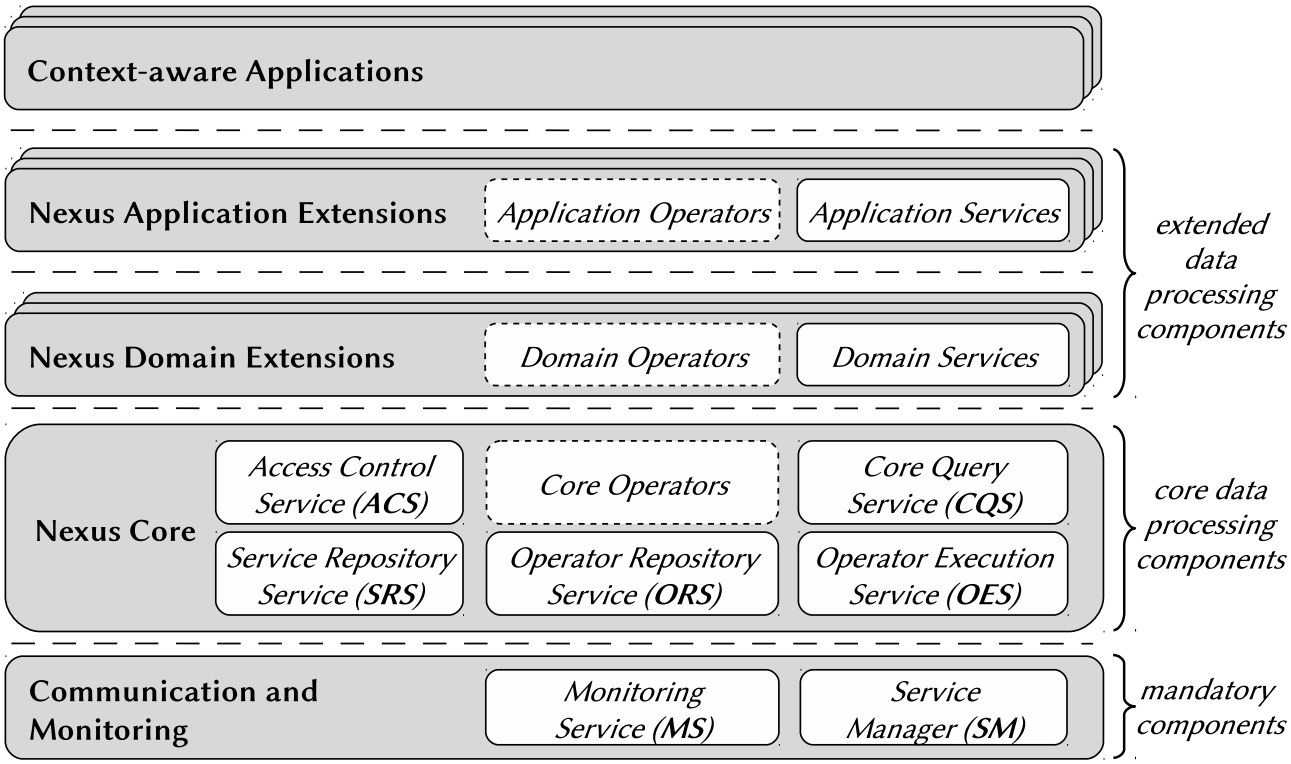


Figure 3.3: NexusDS Layer Architecture

is a distributed stream processing middleware targeting the requirements levied in Section 2.4, thus providing enhanced support for complex application scenarios which require specialized techniques.

The main purpose of NexusDS is to sustain a flexible and seamless extensibility of the system by means of application or domain specific functionality. This especially refers to the integration of operational extensions in terms of *operators*, and functional extensions in terms of *services* into NexusDS. NexusDS consists of four layers as depicted in Figure 3.3. Each layer is built upon its underlaying layers. The architecture combines an adaptive service platform and an extensible operator framework. Operators (displayed as dashed boxes) are push-based and are used to integrate custom data-processing into the system by means of the push-paradigm. In contrast, services (displayed as solid boxes) follow a request-response paradigm and are used to implement customized system behavior in terms of service functionality by means of the pull-paradigm. Starting from the bottom of the figure, a brief explanation of the single layers is provided to get an overall picture of the system. Thereby, the lowest layer is mandatory and is part of each NexusDS node providing basic NexusDS functionality. The higher the layers in Figure 3.3, the more specific they are.

3.4.1 Communication and Monitoring Layer

The *Communication and Monitoring Layer* relies on communication and execution primitives. More sophisticated communication mechanisms are defined within this layer. The Communi-

cation and Monitoring Layer inherits the flexibility and scalability of P2P networks [12] and is inspired by the idea of advanced resource handling and monitoring mechanisms from grid computing systems [54, 75]. This layer contains two fundamental services which are part of each NexusDS node: The Monitoring Service (MS) and the Service Manager (SM). They constitute the basic functionality that each NexusDS node has. The Monitoring Service is the component that monitors the system and keeps track of the computing nodes and their specific characteristics as well as the runtime behavior of executed operators. The Service Manager permits service publication and makes services available within the system. Multiple instances of both services—the Service Manager and the Monitoring Service—are created and distributed across multiple NexusDS nodes to increase availability and to reduce possible bottlenecks. An instance of the Monitoring Service is started on each NexusDS node to collect runtime statistics of the respective node. Besides this, the service also provides statistical data by means of historical data.

The Service Manager is described in more detail in Section 3.5.1 and the Monitoring Service with its related infrastructure is shown in Section 3.5.2.

3.4.2 Nexus Core Layer

The *Nexus Core Layer* contains the core components, i. e. operators and services of NexusDS. Services are instantiated and distributed among NexusDS nodes and its life cycle is managed by the Service Manager. Not every participating node provides all kinds of services, however. This can be adjusted to meet certain requirements such as avoiding a single point of failure or restricting service execution to a certain administrative domain. The core components provide fundamental data stream processing functionality of a DSPS. The Core Graph Service accepts a SP graph in NPGM[2] format and takes the necessary actions for SP graph deployment. The Core Graph Service fragments and deploys the SP graph to meet certain QoS requirements. The fragmentation and deployment process is presented in Chapter 6. Each SP graph fragment is deployed on an Operator Execution Service which controls the execution of the operators in the assigned SP graph fragment. Details on the Operator Execution Service are shown in Section 3.5.4. The *Core Operators* represent the basic operators available in NexusDS and correspond to DB operators commonly known, such as *Selection* and *Projection*. These operators—actually the physical operators—are stored in the Operator Repository Service. The Operator Repository Service provides a query interface to query operators. The repository also counts sources and sinks as operators, having either only outputs or inputs. As with operators, services also need a place to be stored which is represented by the Service Repository Service (SRS). Like the Operator Repository Service, the Service Repository Service also stores the service implementations and provides an interface to query services. Finally, the Access Control Service takes care to enforce the access conditions defined in the system. The Access

[2] A NPGM SP graph is a directed graph which consists of interconnected operators, forming a processing pipeline. The NPGM and NEGM SP graphs are described in the next chapter.

Control Service is needed to check if all necessary permissions are valid to start processing of the requested data streams.

Details on the individual components are discussed later in this chapter.

3.4.3 Nexus Domain Extensions Layer

The *Nexus Domain Extension Layer* builds on the Nexus Core Layer and clusters services and operators for a certain domain of interest. There may exist many of these Nexus Domain Extension Layers in parallel, each providing functionality tailored to a particular domain. The idea of the Nexus Domain Extension Layers is to group functionality needed by many applications and share this functionality with them thus avoiding redundancy. For example we can imagine many visualization applications performing e. g. volume visualization. For the domain of volume visualization appropriate operators are implemented and published via the Operator Repository Service mentioned beforehand. Analogously, a query service—we call this service the volume visualization service—is implemented and made available providing a domain specific language which is tailored to this domain. The volume visualization service translates the visual queries to SP graphs which the Core Graph Service can process. The service eliminates unnecessary complexity for the inquirer and helps focusing on relevant facts related to this domain.

3.4.4 Nexus Application Extensions Layer

Analogously to the Nexus Domain Extensions Layer the *Nexus Application Extensions Layer* enables services or operators that are specific for a single application. Thus, parts of the application logic which is not already covered by domain-specific extensions can be outsourced to NexusDS. Consequently, this outsourced functionality is highly application-dependent and not treated as a domain extension. A reason for such an application-specific extension is that a machine the application is currently running on is not able to process certain tasks in a satisfactory manner, thus making outsourcing of the functionality under concern mandatory. Again thinking of the visualization scenario, current mobile clients still have shortcomings regarding processing power and would rapidly cause an overload situation. E. g., the rendering process is not reasonably executable on a mobile device with limited capabilities. In such a case it is highly recommendable to outsource application-specific parts to a more powerful machine and just receive the final outcome. In this particular case, an additional encoding operator which takes the rendered pictures from the rendering operator, transforms them into an image and provides a video stream of scaled and compressed images (in order to save bandwidth) rather than the full-sized results.

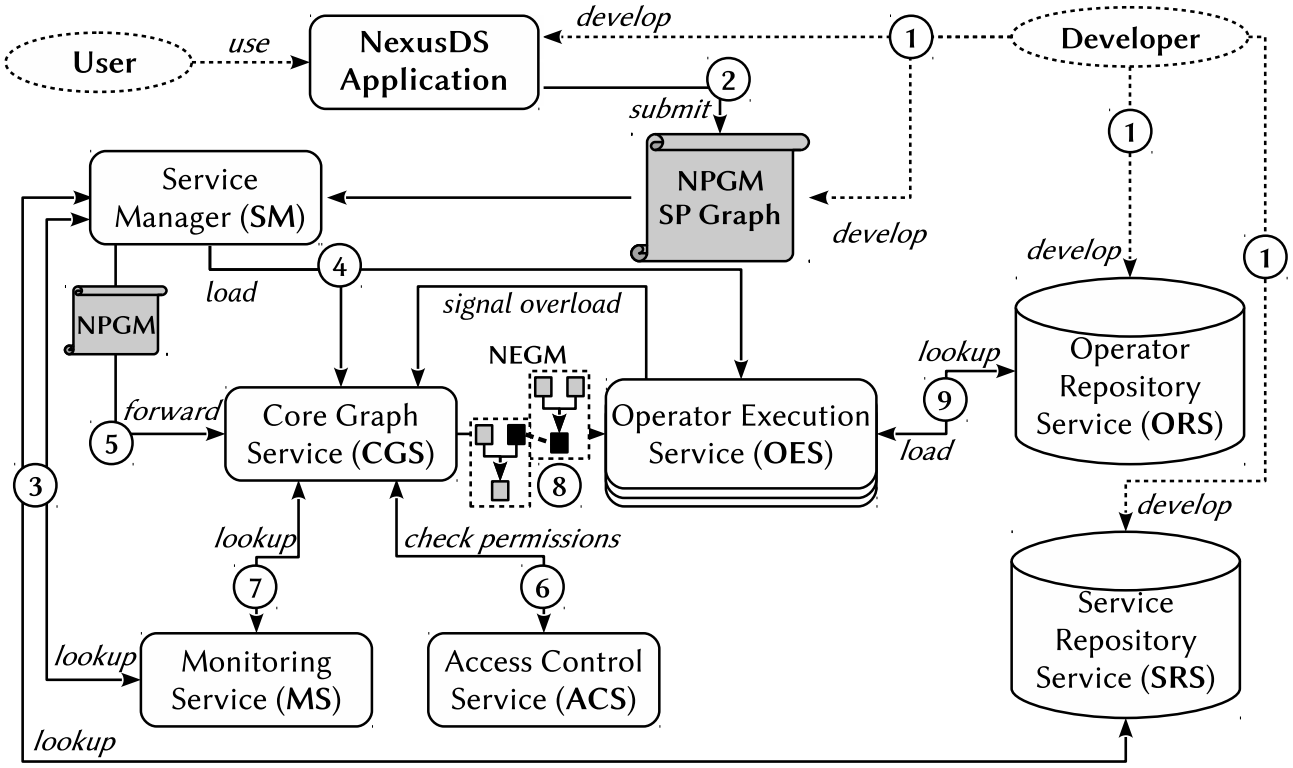


Figure 3.4: NexusDS Components Architecture

3.4.5 Context-aware Applications Layer

The top layer is represented by the *context-aware applications* that exploit the NexusDS system to process their data. Examples for such applications have been presented in Section 2.3.

3.5 NexusDS Components Architecture

The subsequent explanation of the NexusDS components architecture is supported by Figure 3.4. It shows the interrelations of the services in NexusDS. Firstly, the deployment of SP graphs is presented. Then, the single components are detailed in their respective sub sections.

The processing model basically consists of three phases. In the first phase, a developer must implement the application, the operators, the SP graph—which in our case is a NPGM SP graph—and possibly specific services. In the second phase the application sends an SP graph to NexusDS. Finally, in the third phase, the SP graph is deployed and processing starts.

First, a *developer* (usually a domain expert) develops the application, the domain extensions (operators and services) and typically also formulates the SP graph (1). The domain extensions may vary according to the specific requirements. For the following, the sample scenario of a location-aware visualization pipeline from Section 2.3.1 is assumed. In this case, the specific requirements are described by the domain-specific operators *Render*, *Map*. But the development

could also comprise the development of a domain-specific service offering a domain specific language. Then the developer creates a client application to receive the resulting data. The received data can then be arbitrarily processed by the internal application components. The application developer must not necessarily be the same person as the domain-expert. However, the application exploits the domain-specific components to implement its actual functionality. The location-aware visualization application submits an NPGM SP graph to the Service Manager (2). As described in Section 4.1.2, the *user* and also the *application* itself can specify constraints. E. g., the user could indicate preferring the resulting image stream in a certain resolution. The application could constrain the processing of data to only Operator Execution Services considered secure. The Service Manager looks up the Service Repository Service for services able to process the received document and thereafter checks their load state at the Monitoring Service (3). If no service is currently available, the Service Manager loads the corresponding services (4). The Service Manager manages the life cycle of the services in NexusDS. He then forwards the SP graph to the Core Graph Service (5).

The Core Graph Service transforms the NPGM SP graph to an equivalent NEGM representation, i. e. a deployable representation of the original NPGM SP graph. The Core Graph Service first checks if the user has the necessary permission to access and process data (6). Furthermore, all components involved in data processing, i. e. services, NexusDS nodes, operators etc. must have the necessary permission. Also, the Core Graph Service verifies if the *user relevant constraints*, *application relevant constraints*, and *domain relevant constraints* match the *system relevant constraints*. Therefore, the Core Graph Service looks for available Operator Execution Services at the Monitoring Service which satisfy the constraints and to which distribute the NPGM SP graph on (7). I. e. the Operator Execution Services must comply with the defined deployment constraints as well as runtime constraints. The algorithm for SP graph distribution is detailed in Chapter 6. The NEGM SP graph is deployed to the Operator Execution Services (8).

At this point, the NEGM SP graph is ready for execution. But before execution starts, the Operator Execution Services looks up the physical operators necessary from the Operator Repository Service. These operators are then loaded and configured according to the received NEGM SP graph (9).

Finally, SP graph execution starts, producing a continuous data stream. The resulting continuous data stream is sent to the application which then processes it accordingly.

Next, the main components of the NexusDS system are presented in more detail. The order of description mostly corresponds with the order in which these components appear in the processing sequence presented above.

3.5.1 Service Manager

The SM manages the life cycle of services. The architecture of the Service Manager is depicted in Figure 3.5 and basically consists of two areas: A Service Manager Interface (SMI) which receives query documents and—depending on internal state—eventually delegates its processing and a Service Management Area (SMA), providing an area to load services (displayed on the left side) and facilities for instantiating services on the NexusDS node (displayed at the bottom).

The Service Management Area provides an environment for service execution. Thereby it offers an interface to receive incoming query documents. It also provides an interface for the Service Loader (SL) to start, stop, or pause the execution of a service instance. This component provides a life-cycle management for services. Finally, the Service Management Area offers an interface for instantiating services.

In order to load and start services on a NexusDS node, the Service Loader looks up and retrieves service package binaries from the Service Repository Service. The Service Loader also holds services already retrieved in a local cache, instantiates the services in the Service Management Area (if necessary) and removes services which are no longer needed. The Service Loader furthermore provides a small registry of all service instances running on the local node. This is needed for the Service Manager Interface to answer the question if a service already running on this NexusDS node is capable of processing the query document. The answer to this question is useful for the Service Manager Interface when deciding whether to process a query document locally or not.

Once services have been instantiated and started, they are ready to process incoming documents. During the instantiation process, each service also publishes its specific connection information. This service-specific connection information is needed for *direct service invocation* as described in Section 3.5.1.1. The direct invocation mechanism is useful if a specific service instance should receive the query document. The default way is to send the query document to the Service Manager, which then handles the invocation of the correct service transparently, either by a locally running instance or a remote one.

The decision whether a local or a remote service instance should process the query document is taken by the Service Manager Interface. The decision can be made depending on different criteria, such as the current load on the NexusDS node or if a service is already running, offering the requested service. In this sense, the Service Manager Interface acts as a component which makes its routing decisions, depending on the local load situation and contents of the query document. The internal functioning of this component is similar to the idea of an Enterprise Service Bus (ESB) [33]. The ESB provides an architectural approach for service and application integration in enterprises. These enterprises usually have a heterogeneous service infrastructure. The ESB approach tightens the communication between those loosely coupled and heterogeneous services by a shared communication bus. However, point-to-point connections between different services are still possible. The Service Manager Interface works in a similar way: It represents a shared communication bus for services and applications of

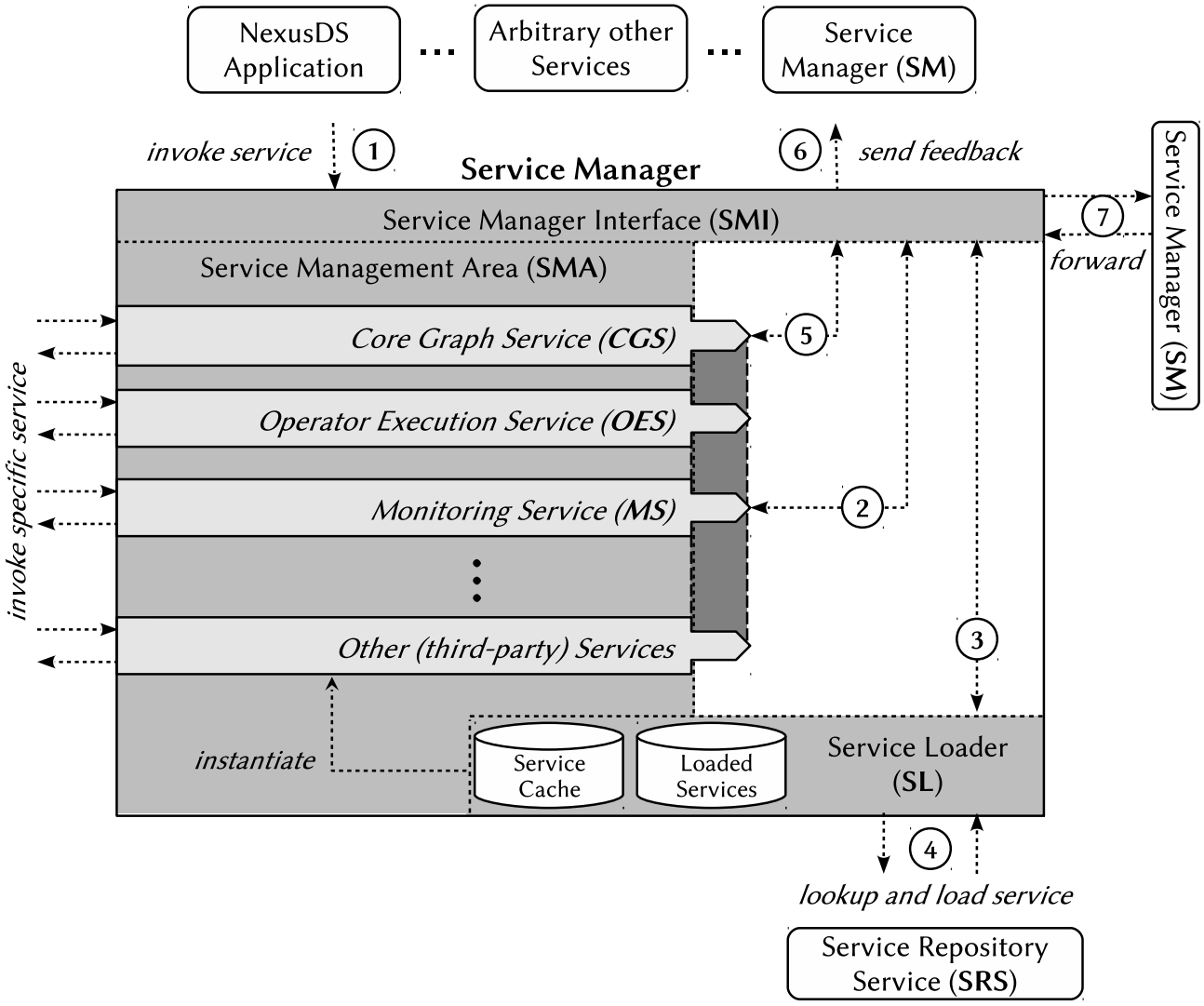


Figure 3.5: Architecture of the Service Manager (SM)

NexusDS and routes query documents (as well as the corresponding resulting documents) depending on the local load situation and on its contents.

When an inquirer invokes the Service Manager by sending a query document to it, the following steps are performed by the Service Manager (as shown in Figure 3.5): The query document, having a specific document format, is received by the Service Manager Interface (1). The Service Manager Interface needs to decide whether the query document is processed locally or must be forwarded to a remote Service Manager-instance. The Service Manager Interface first checks if the local workload allows a local processing of the query document, i. e. by asking the local Monitoring Service-instance (2). If the result is positive, i. e. no overload situation is detected, it is checked if the service necessary for query document processing is already loaded and running on the local node (3). If the service instance is not locally available, a lookup in the Service Repository Service is needed (4). The Service Repository Service returns the service binaries. The binaries are locally stored in a *service cache* and the *loaded service*

is also registered locally after instantiation. Moreover, the instantiated services also register to the Service Repository Service in order to be discoverable and directly accessible (see Section 3.5.1.1). This last step is necessary since something might go wrong when instantiating the service. Once the requested service is locally available, the original query document is passed on to the service instance (5). The service instance then processes the query document and produces a feedback or resulting document which is sent back to the inquirer (6). After the resulting document has been sent to the inquirer, the request has been successfully processed. However, the Service Manager Interface might decide to forward the query document to an alternative Service Manager instance running on a different NexusDS node. This might happen if e. g. the load situation prohibits the local processing of the query document (7). In this case the remote Service Manager performs steps (1) to (7), as described above, instead of the Service Manager the document was originally sent to. It is important to note that the inquirer always receives the resulting document by the original Service Manager instance the query document was sent to. This is because the interaction between the inquirer and the service being invoked is synchronous. The same also holds for inter-service communication.

3.5.1.1 Direct Service Invocation

All services in NexusDS have the same interface. This applies in particular to the SM, which however is a special service managing the life cycle of services. Nevertheless, the interface used for communication is the same as in other services. From this point of view all services are similar, although they differ in their implementation details. Besides applications, services can send query documents to (other) services when necessary. This means that beside service invocation using the Service Manager, applications and services can also directly invoke other services. This is useful if a certain service instance on a specific NexusDS node should receive the query document instead of (from an inquirer's point of view) an arbitrary service instance. Each time the Service Manager Interface instantiates a service, the service also registers itself at a globally accessible *service registry*, the Service Repository Service. Thus, an application or services can query the Service Repository Service to get the endpoint information of a specific service instance and directly send a query document.

Two basic communication principles are possible, *indirect communication* and *direct communication*. The indirect communication is performed by sending the query document to an arbitrary Service Manager instance running on a NexusDS node. The Service Manager delivers the document to the correct endpoint. Alternatively, the *direct communication* is performed by querying the Service Repository Service about available service instances of interest, directly sending the document to them. This communication mechanisms allows direct interaction with a specific service instance and implements a loosely coupled interconnection of services according to the SOA principle described in Section 2.5.6.

An example for the direct communication pattern is between the Core Graph Service and the Operator Execution Service. The Core Graph Service receives a NPGM SP graph which must be transformed into an NEGM SP graph. The NEGM SP graph is then directly sent to the

Operator Execution Service instances involved in the processing. Thus, both communication mechanisms are needed in order to make the services work properly. Other communication patterns are also possible, mainly depending on the actual service.

3.5.2 Monitoring Service

Lastly, the Monitoring Service (MS) observes the activities of the Operator Execution Service instances running on NexusDS nodes, and provides the collected information on demand. The Monitoring Service represents a component which might be implemented as a distributed service but appears to the outside as being a monolithic one. Many different patterns might be used to realize this service, all of them with their particular characteristics: An actually centralized component (e. g. a DB server), a purely distributed shared-nothing component (e. g. each service instance holds its own data), or a hybrid shared-memory component (e. g. building a kind of super-peer overlay). For the following discussion, we will desist from this fact and present the conceptual idea of this component instead.

The task of Monitoring Service is not solely limited to collecting and providing statistical as well as status information on NexusDS nodes. As depicted in Figure 3.6, the Monitoring Service also offers value-added functionalities by firstly preprocessing the collected statistical data, and secondly providing an interface to filter NexusDS nodes which do not meet SP graph requirements[3]. The analysis of NexusDS nodes capable of executing a certain operator by satisfying certain constraints at the same time is mandatory and should possibly be done where the data resides to avoid transfer of high data volumes.

For this purpose, the Monitoring Service accepts NPGM SP graphs as originally sent to the Core Graph Service. The NPGM SP graph provides the processing pipeline definition on how to process the data. Furthermore, it defines QoS-related information on how the SP graph should be deployed and executed. The details about QoS specifications are presented in Chapter 6. For the moment it suffices to know that QoS-specific statistics are collected and that the QoS specifications of a SP graph define how SP graphs should be deployed and executed. After processing the NPGM SP graph, the Monitoring Service returns a modified version of the original SP graph. The resulting SP graph is augmented with lists of NexusDS nodes potentially capable of executing the respective operators. Thus, this process prunes the search space and limits the number of NexusDS nodes to consider for SP graph deployment (see Chapter 6).

To accomplish this task, the incoming measurement data is preprocessed and provided for further operations. For this purpose, the Monitoring Service is subdivided into two parts: A *measurement data processing* part and a *NPGM SP graph processing* part. The measurement data processing part receives incoming measurement data originating from the Statistics Col-

[3]Recall: As explained in the former chapter, operators might have requirements which must be met by NexusDS nodes in order to execute these operators. If we think of a visualization scenario, the rendering of a scenery would be such an operator. We call these operator-related restrictions *constraints* which are described in more detail in Chapter 4.

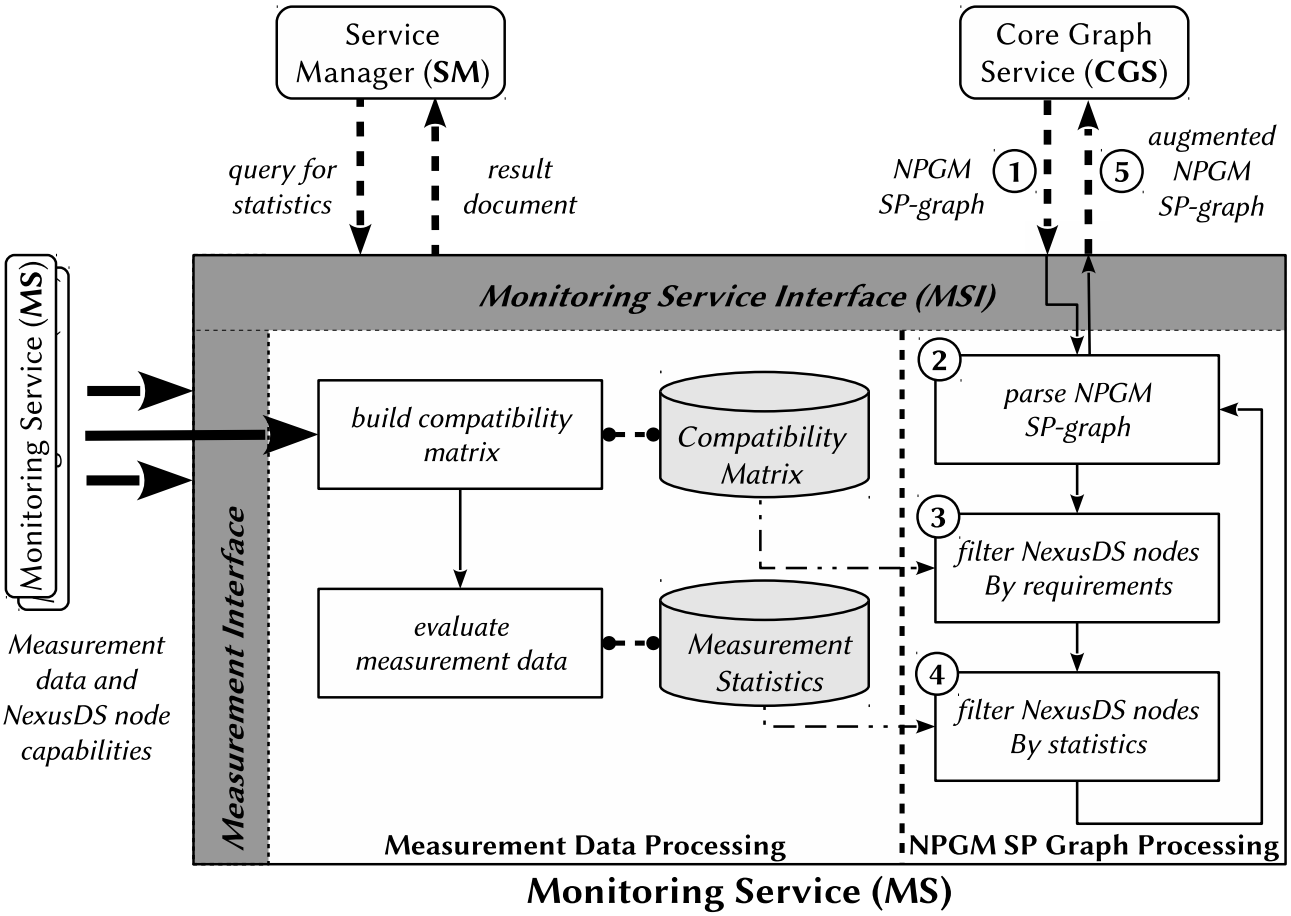


Figure 3.6: Architecture of the Monitoring Service (MS)

lector Client which is part of the Operator Execution Service. However, measurement data might also originate from other Monitoring Service service instances, e. g. in a shared-memory approach scenario (as mentioned above).

At first *compatibility matrix* that matches the capabilities of each NexusDS node with the requirements of the operators is created. The capabilities of the NexusDS nodes are collected. As mentioned in Section 3.2, operators might have specialized component requirements such as an installed GPU in order to run properly. On the other hand, NexusDS nodes have certain capabilities. These NexusDS node capabilities are collected and published by the Operator Execution Service (OES) instances running on the corresponding NexusDS nodes. The compatibility matrix stores all combinations of operators and NexusDS nodes. Additional combinations can be added incrementally to the compatibility matrix, as new operators are created and new NexusDS nodes emerge. This step is needed to filter incompatible NexusDS nodes from the set of NexusDS nodes.

Then *measurement statistics* are created by evaluating the received measurement data of the NexusDS nodes. The measurements include the current NexusDS node status and the runtime measurements of the operators currently being executed on. This data is sent to the

Monitoring Service in equidistant time intervals. The Monitoring Service then creates the different statistical data partitions. There are three different types of statistics:

- **node statistics (N statistics)**: For each NexusDS node these statistics represent its current state. The status is delivered along with the remaining statistical measurements that are sent to the Monitoring Service.
- **node-operator statistics (NO statistics)**: The performance of an operator mainly depends on two aspects: Its parametrization and the NexusDS node it is executed on. NO statistics provide the performance of each operator on a specific NexusDS node with a specific parametrization.

  However, storing the NO statistics separately for each parametrization seems not to be feasible. Determining performance-relevant parameters is a non-trivial task and error prone. Instead, the operator developer indicates the performance-relevant parameters having a massive influence on the runtime behavior of an operator. For these parameters, the similarities are calculated. For numerical values the average sum of all distances between the single parameter values are calculated. For non-numerical values this is a binary decision, i.e., equality. This means, for non-numerical values all relevant parameters must be equal.
- **node-node statistics (NN statistics)**: These statistics are necessary in order to get information on how NexusDS nodes are interconnected. In practice, these statistics include typical network related QoS metrics such as latency and bandwidth. These statistics are not needed by the Monitoring Service to filter the available NexusDS nodes. The NN statistics are evaluated by the deployment algorithm multi-target operator placement (M-TOP) presented in detail in Chapter 6.

The respective statistical data consists of a fixed length time series of the corresponding QoS target, representing a characteristic diagram and the expected probabilities for a certain value as well as the minimum, maximum, and average values.

The augmentation of a SP graph with compatible NexusDS nodes for each operator works as depicted in Figure 3.6. Firstly, the received SP graph is analyzed and the contained operators are determined (1). The result is a list of operators for which compatible NexusDS nodes are to be determined for execution. This list is synchronized with the compatibility matrix, resulting in a set of NexusDS nodes attached as a candidate list to each operator (2). This candidate list is further refined in the next step (3). For all operators the candidate list is traversed and the NexusDS nodes contained are filtered according to their current state, i. e. when a specific NexusDS node already has an above-average load. Afterwards, the statistical measurements are considered for the remaining NexusDS nodes. The NexusDS node specific characteristics are used to further filter the candidate list, so NexusDS nodes are discarded which do not satisfy the requirements of an operator (e. g. available main memory). Statistics for operator execution mainly depend on two aspects: *Operator parametrization* and the *NexusDS node* a specific operator is executed on. These characteristics are particular for each operator-node combination, such as *processed data items per time unit*. The resulting SP graph with the

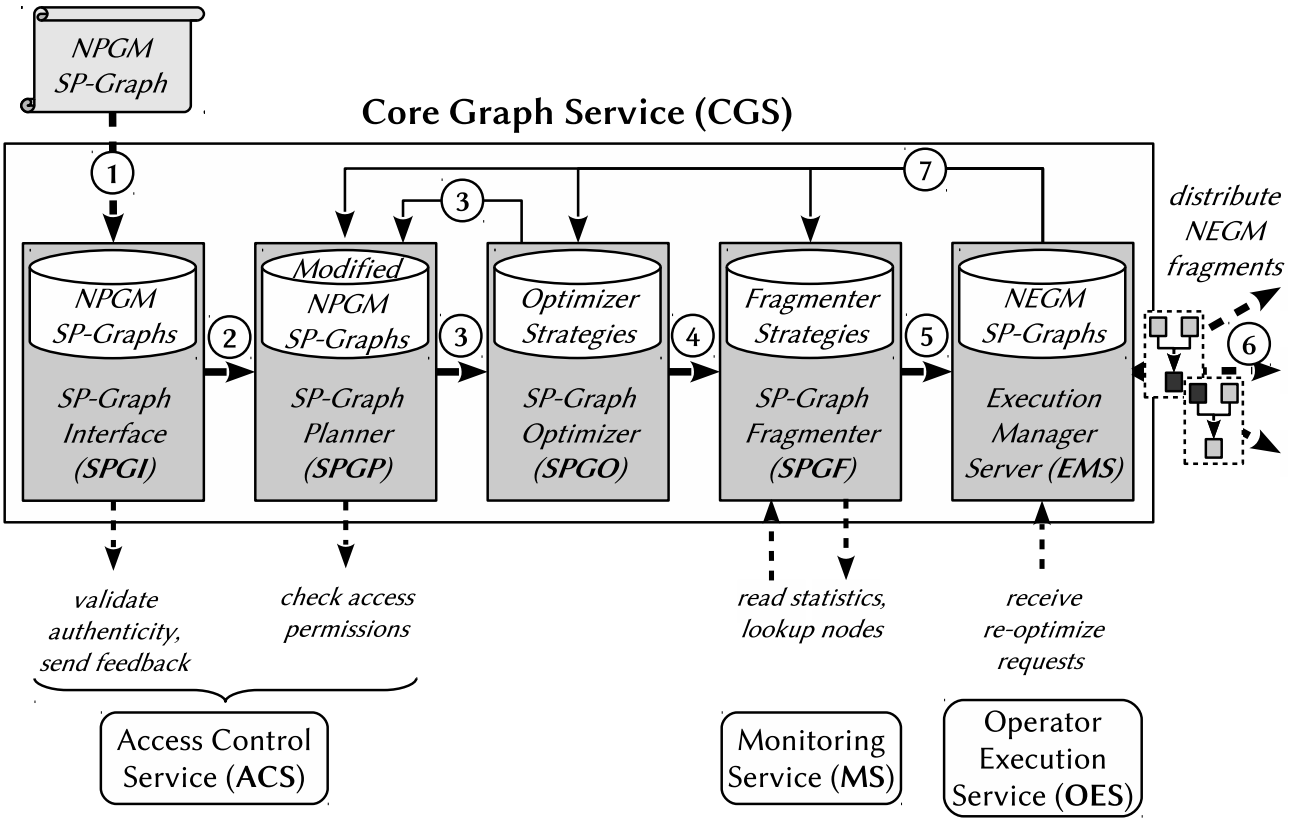


Figure 3.7: Architecture of the Core Graph Service (CGS)

candidate lists attached for each operator is returned. Finally, this modified SP graph is used by the Core Graph Service (CGS) for the deployment process, which is presented in detail in Chapter 6.

3.5.3 Core Graph Service

The Core Graph Service accepts a NPGM SP graph and processes it to get a NEGM representation of the original NPGM SP graph that is deployable and executed in the Operator Execution Service. The architecture of the Core Graph Service is depicted in Figure 3.7 which is described in more detail as follows.

NPGM SP graphs are received through the SP-Graph Interface (SPGI) (1) which offers a connection interface to applications and other services. After receiving a NPGM SP graph, the SP-Graph Interface verifies the inquirer's authenticity by querying the Identity Administration Point, as part of the Access Control Service. If verification fails, an error message is returned. If verification succeeds, the SP-Graph Interface stores the NPGM SP graphs received in a local SP graph cache (denoted by *NPGM SP graphs*) for potential restructuring tasks. The NPGM SP graph is signed by the Core Graph Service and assigned an NPGM ID which is sent back to the inquirer and uniquely identifies this NPGM SP graph among others. With this NPGM ID the inquirer is able to modify or stop execution of the NPGM SP graph.

The NPGM SP graph is then passed to the SP-Graph Planner (SPGP) (2). This component is responsible for initiating the augmentation of the NPGM SP graph with relevant security policies. The augmentation process is performed by the Policy Decision Point, as part od the Access Control Service. After the augmentation, the SP-Graph Planner works in two steps. In the first step the SP-Graph Planner checks whether the inquirer submitting the NPGM SP graph is allowed to execute all operators involved and access the referenced data. In a second step it evaluates if the interconnected operators are allowed to be executed in this order and if all subsequent operators have reading permission for data produced by previous operators. This is necessary to ensure that the processing pipeline can be deployed and executed successfully. However, this is not a final security guarantee since the policies might also depend on the computing nodes the operators are allowed to be executed on. In principle, it may happen that no suitable operator deployment can be found. The solution of the security policies is described in more detail in Section 5.4.

Thereafter, the NPGM SP graph is received by the SP-Graph Optimizer (SPGO) (3) performing a logical NPGM SP graph optimization. A logical optimization operation may be to push selective operators back to the sources to reduce data volumes. This step basically consists of rules which have been proven to produce valid and more efficient SP graphs. As part of this thesis, we did not further investigate optimization rules of NPGM SP graphs as the main focus is getting an executable representation of NPGM SP graphs. Thus, the SP graph is left unmodified and only a physical optimization (in terms of SP graph transformation from an NPGM to a NEGM representation) is performed. The elaboration of this component is left as a future work.

In step (4) the NPGM SP graph is forwarded to the SP-Graph Fragmenter (SPGF) optimizing the physical NPGM SP graph in order to find a deployable and executable representation of the NPGM SP graph. The SP-Graph Fragmenter looks for suitable data sources and for physical operators (stored in the Operator Repository Service) to the corresponding logical ones defined in the NPGM SP graph. Finally, the SP-Graph Fragmenter has to search for suitable participating nodes running an instance of the Operator Execution Service, capable of running the selected physical operators. Once all this information is available, the SP-Graph Fragmenter finds a valid (and possibly an optimal) NEGM SP graph representation of the original NPGM SP graph. The NEGM SP graph is fragmented according to the selected participating nodes and *Fragmenter Strategies*. To support a flexible fragmentation process with additional fragmentation strategies, additional definitions on how to fragment the NPGM SP graph might be added. The default fragmentation strategy is described in Chapter 6.

Finally, the Execution Manager Server (EMS) receives the fragmented NEGM SP graph (5) ready for deployment. The Execution Manager Server stores the fragmented NPGM SP graph in a local cache (associated to its NPGM ID) and distributes the single fragments to the target Operator Execution Service (6) by keeping in sync with the involved Operator Execution Service instances. This is crucial since in general system conditions may vary over time, making it a necessity to re-organize the NEGM SP graph (7). The re-organization is initiated either

periodically or on demand. The periodic reorganization avoids critical situations such as a node overload by checking for overload indicators with support from the Monitoring Service. On-demand reorganization is initiated by one of the Operator Execution Service instances involved, usually if an overload situation is detected which cannot be handled by the Operator Execution Service. In this case the Operator Execution Service instance signals to the associated Core Graph Service (via the Execution Manager Server) to eventually re-organize the NEGM SP graph. Reorganization starts with the SP-Graph Planner, SP-Graph Optimizer, or SP-Graph Fragmenter. E. g., if a security policy has changed, the reorganization process starts with the SP-Graph Planner. If an operator needs to be parallelized, re-organization tasks on a fragment basis can be performed by the SP-Graph Fragmenter. For reorganization, either *load-shedding* techniques [58, 93, 132, 140] or *operator migration* tasks are available options for SP graph modifications [100, 151, 155]. The modified NEGM SP graph fragments are then sent to the affected Operator Execution Service.

3.5.4 Operator Execution Service

After the distribution of the NEGM SP graph fragments to the respective Operator Execution Service (OES), each Operator Execution Service instance instantiates the operators contained in the NEGM SP graph fragment and starts execution. The architecture of the Operator Execution Service is shown in Figure 3.8. Its main components are the Operator Execution Environment (OEE), providing an environment for operator execution and its supporting units Statistics Collector Client (SCC) to collect runtime statistics, Operator Scheduler (OpS) to do a re-scheduling of locally running operators, Operator Repository Client (ORC) handling locally missing operators, and finally Execution Manager Client (EMC) to receive the NEGM SP graph fragment and signal overload situations. The basic working method of the Operator Execution Service consists of four phases: An *initialization* phase, an *execution* phase, a *lightweight adaptation* phase, a *heavyweight adaptation* phase, and a *termination* phase. In the following, the single phases of the Operator Execution Service are described in more detail.

Initialization: The EMC receives an NEGM SP graph fragment (1), which defines how this Operator Execution Service instance is going to process data. The first step consists of storing the NEGM SP graph fragment and its associated NPGM ID into a local *NEGM Fragment* cache to keep track of all fragments currently being executed on this Operator Execution Service instance. Then, the Operator Execution Service must check whether all operators needed to execute the NEGM SP graph fragment are locally available. The local *operator cache* must be checked by querying the ORC (2). If the operator is locally missing, the Operator Repository Client contacts the Operator Repository Service (ORS) asking for the executable code of the missing operator. If the operator is known and the computing node the OES instance is being executed on is allowed to access the executable code, the operator package is downloaded by the ORC and stored in the local operator cache for later reuse. Once all operators are locally available, an instance of all relevant operators is created. The operator instances are

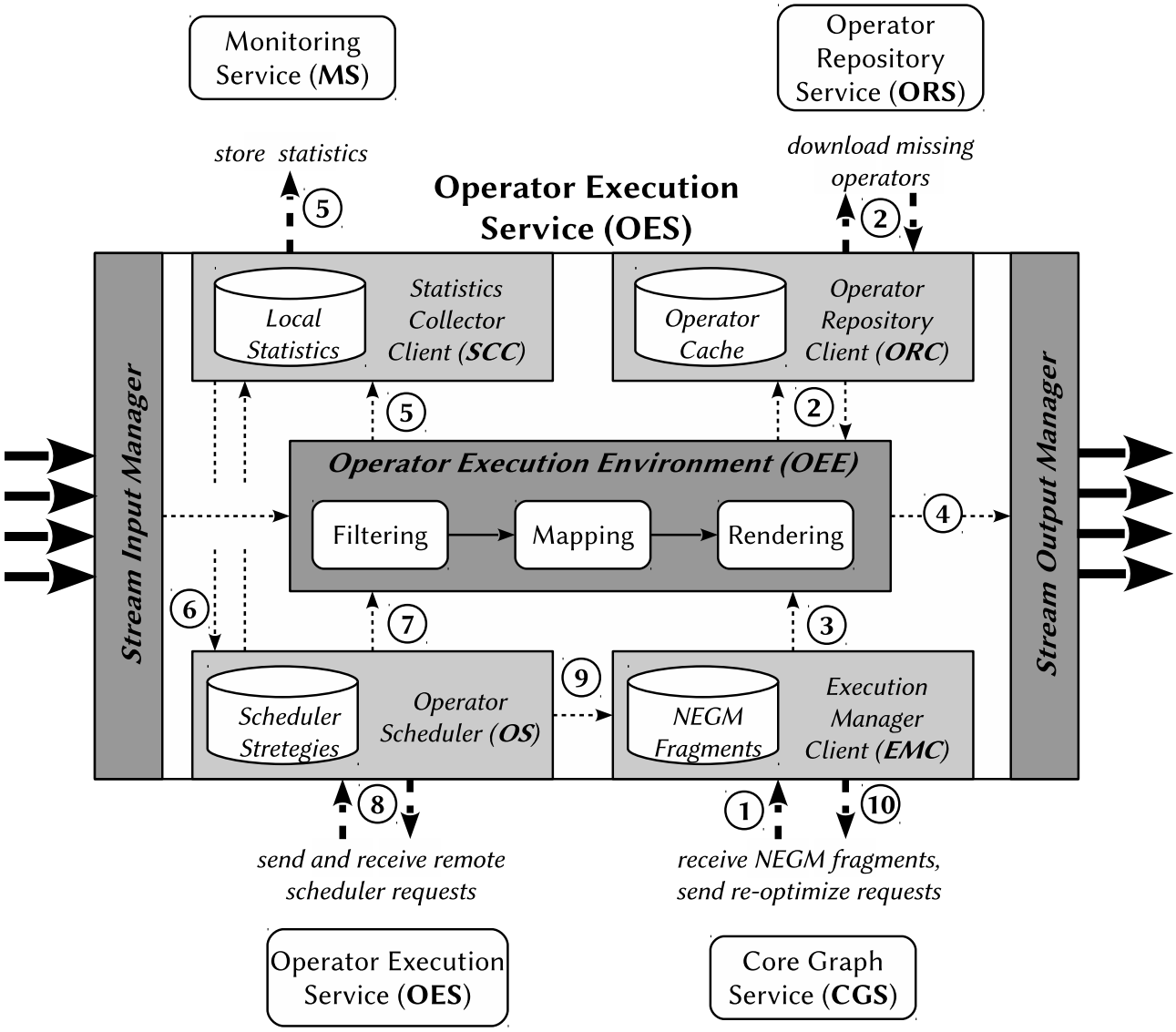


Figure 3.8: Architecture of the Operator Execution Service (OES)

parametrized and interconnected according to the NEGM SP graph fragment definition, and finally executed within the Operator Execution Environment (3).

Execution: The Operator Execution Service at this point waits for incoming data to be processed. Incoming data is handled by the *Stream Input Manager* and is routed to corresponding operator inputs. The operators of the NEGM SP graph fragment process incoming data and generate output (4) which is sent to the *Stream Output Manager*. The Stream Output manager forwards the processed data to subsequent Operator Execution Service instances on remote participating nodes for further processing. During operator execution, statistics are collected by the Statistics Collector Client. Runtime statistics contain information such as bandwidth consumption and available bandwidth, memory-footprint of the single operators or processed items per time unit. The statistics are collected depending on the actual parametrization of the operator, since the parameter configuration usually has a great influence on the runtime

behavior. The statistics are stored on the participating node running the Operator Execution Service instance and are also transmitted to the Monitoring Service making the statistics available to other NexusDS components (5). Beside the statistics, the respective operator and node information are also transmitted in order to uniquely identify a runtime statistics record. This is important since the runtime behavior of operators heavily depends on three factors: The operator implementation itself, the parametrization of the operator, and the node it is executed on. These runtime statistics are used besides others by the Operator Scheduler to schedule operator execution. Changing conditions can make it necessary to adapt NEGM SP graph fragments at runtime. For that two options are available: Either a *lightweight adaptation* or *heavyweight adaptation*.

Lightweight Adaptation: *Lightweight adaptation* refers to actions that are limited by computation node borders and are only locally applicable. In contrast to this, the *heavyweight adaptation* refers to actions that exceed computation node borders and involve other services (described below). During operator execution it may be necessary to change the execution order within the processing pipeline [17, 32, 142]. The Operator Scheduler organizes the order in which operators are executed within the Operator Execution Service by using the collected runtime statistics of the Statistics Collector Client (6). The lightweight adaptation process is performed until the local adaptation potential for an Operator Scheduler instance is exhausted. This occurs if the arrival rate of data elements is higher than the Operator Execution Service (and accordingly the operators) can handle or by a computation node failure. In this case, the Operator Scheduler has two options. The first option is to change the operator scheduling strategy (7). For this purpose Sutherland et al. [130] proposed a flexible operator scheduling framework that adjusts the currently active scheduling strategy to maximize a given QoS. The second option is to ask preceding Operator Execution Services (by contacting their respective Operator Scheduler) to slow down processing speed. This can be achieved by either changing the scheduling strategy or implementing a *chain-scheduling* strategy (8). Alternatively, modifying the operator parameters is conceivable, which however is error-prone and complex. Furthermore, for this the Operator Scheduler needs to know about the semantics of each operator and each of its parameters, which is difficult to implement with the flexible operator framework supported by NexusDS.

Heavyweight Adaptation: When *lightweight adaptation* is not applicable or even fails, the consequence is a *heavyweight adaptation*. Heavyweight adaptation is initiated by the Operator Scheduler when scheduling is not enough to avoid critical situations such as buffer overflows. If no suitable precautions are taken, incomplete output data or even worse node failures may be the result. This requires techniques as described in [5, 19, 131]. To initiate heavyweight adaptation, the Operator Scheduler signals the Execution Manager Client that the current NEGM SP graph fragment execution must be modified (9). The Execution Manager Client itself contacts the Execution Manager Server of the Core Graph Service (see Figure 3.7) to initiate the heavyweight adaptation process (10) signaling that the NEGM SP graph must be restructured. The Core Graph Service performs the NEGM SP graph modification process as described in Sec-

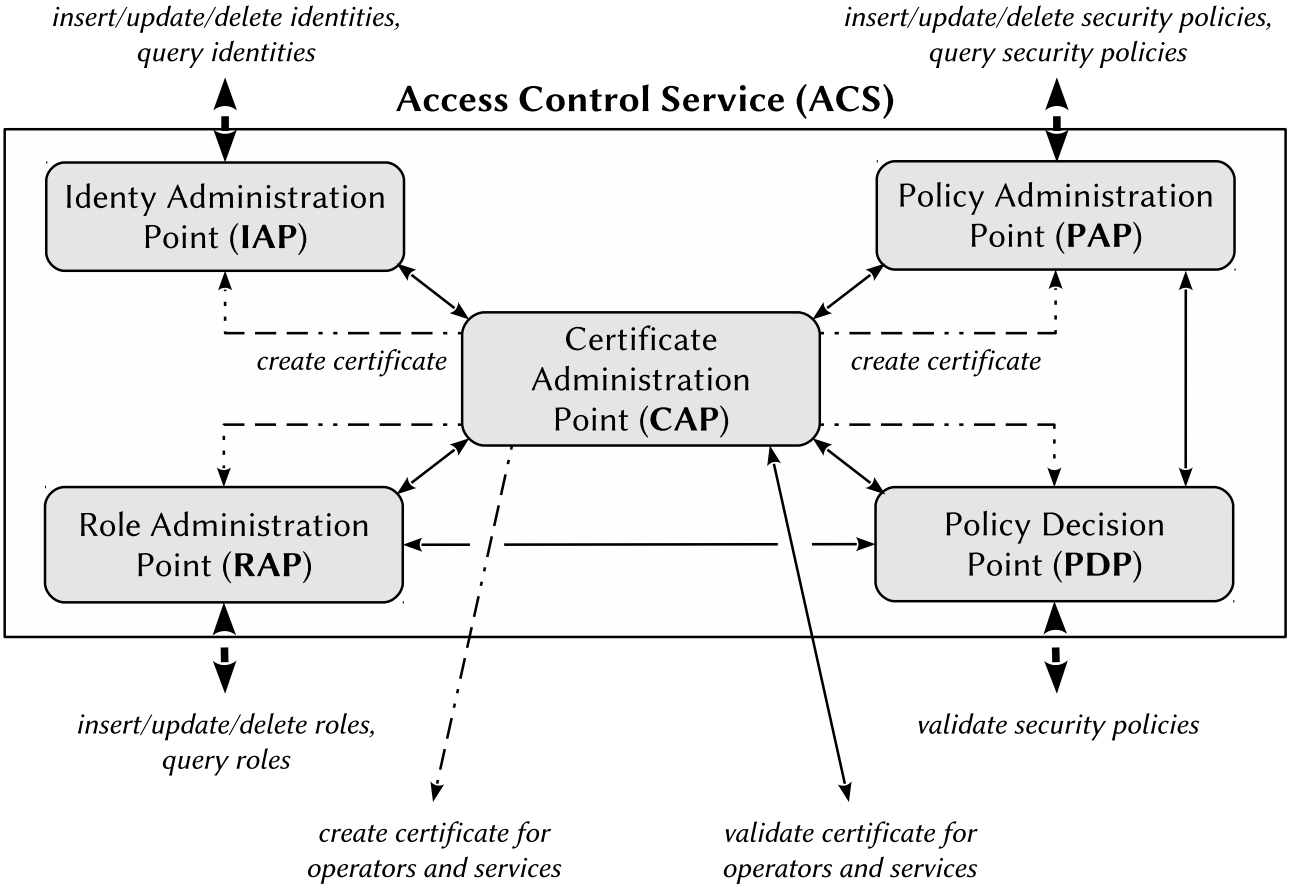


Figure 3.9: Components contained in the Access Control Service (ACS)

tion 3.5.3. The modified NEGM SP graph fragments are then received back and the Operator Execution Service performs necessary steps for modified SP graph execution.

Termination: The execution of the NEGM SP graph fragments ends when the lifetime of the associated NPGM SP graph expires or when the inquiring client actively terminates the running NPGM SP graph using the received NPGM ID at SP graph deployment time. The client sends this NPGM ID to the Core Graph Service which in turn initiates the *termination phase*, propagating a termination signal to the involved Operator Execution Service instances.

3.5.5 Access Control Service

The Access Control Service as depicted in Figure 3.9 consists of a set of services. The Access Control Service allows to define and validate access, process, and granularity control policies on data. In the following, the Access Control Service components are described in more detail.

A central component of the Access Control Service is the Certificate Administration Point (CAP) which is responsible for creating certificates for and verifying certificates of services and operators. All Access Control Service related services and more generally all components interacting with the security infrastructure need a valid certificate to identify themselves as

trusted components. These certificates uniquely identify entities. The certificates however are not mandatory for the components to work with NexusDS. They can also run their routines without certificates unless they are meant to be executed in a secured environment. In order to interact with the Certificate Administration Point an interface is provided to create and validate certificates.

The Identity Administration Point (IAP) registers entities and lists all entities known to the security framework. Entities are denoted *subjects* in the following. A subject might be an application, a user, or a service asking to interact with subjects of restricted areas or demanding access to data within the restricted service. The Identity Administration Point utilizes the well-known concept of a combined *user name* and *password* to uniquely identify subjects. The related certificates are not stored here. Instead, each time a subject queries the Identity Administration Point a valid certificate must be provided by the inquirer. The Identity Administration Point provides an interface to query, insert, update and delete identities. Participating nodes (nodes running, e. g. the Operator Execution Service) are also subjects and thus need to identify themselves to the Identity Administration Point. This is due to the fact that they might execute operators or run services which process restricted data themselves or are only allowed to work in the restricted area.

The Role Administration Point (RAP) performs role management according to the Role Based Access Control (RBAC) model [8] to group different identities depending on their function within the system. The security policies are associated to roles instead of connecting them to a concrete identity because usually a great number of users participate in such a system. Verifying each identity and assigning the security policy for each of them is cumbersome and error prone. By grouping them a semantic interpretation is provided and furthermore security policy assignment is highly facilitated. This means that each role derived from another role inherits all related security policies. Inherited security policies cannot be removed but new ones might be arbitrarily added assuming these are not in conflict with each other.

Finally, the Policy Administration Point (PAP) and the Policy Decision Point (PDP) represent the components which are responsible for security policy management and security policy evaluation. The Policy Administration Point manages security policies and provides an interface for adding, modifying and querying security policies. These policies in association with the associated rules are exploited by the Policy Decision Point to decide whether a subject is allowed to perform the requested action. The Policy Decision Point plays an essential role for the Core Graph Service (see next section) before mapping a (logical) SP graph to an executable representation. To check if the security policies deposited allow the execution by the inquirer, the Policy Decision Point receives a SP graph, collects all relevant security policies from the Policy Administration Point and attaches them to the SP graph. Depending on the selected security level, the Policy Decision Point checks if all components involved—operators, computing nodes, inquirer and so forth—own the necessary security policies and possess valid certificates. The augmented SP graph is returned to perform the SP graph validation within the Core Graph Service.

3.5.6 Core Operators, Operator Repository, and Service Repository

Core operators in NexusDS include operators of the Augmented World Query Language (AWQL) presented in Section 2.5.2.2. The two operators *Selection* and *Projection* are provided by AWQL which have the same functionality as the equivalent operators in a DBMS. The combination of different data sets (e. g. by *Join* operators) is not supported by AWQL.

The main focus of this thesis is not on the development of physical operators but rather to provide the necessary infrastructure that satisfies the requirements from Section 2.4 and supports the integration of operators—as well as services—in a seamless way. A result of this is the operator model which is described in detail in the next chapter. Some core operators have been implemented which helped to develop the sample applications presented in Chapter 7. These operators are *Selection* to filter data according to a given predicate and the Spatial Model Server (SpaSe) source operator. This source operator accepts an AWQL query and constructs an AWQL cursor query. The resulting AWQL cursor query is forwarded to the Nexus system. Instead of sending back the complete integrated result set (which may be composed of results from multiple data providers), the Nexus system constructs a cursor structure to iteratively retrieve the results, and waits for cursor interaction. The source operator in a second step polls for next data items which are then pushed into the stream processing pipeline for further processing.

In order to make the operators available in NexusDS, a storage area to insert, update, delete and query these operators is needed. For that purpose the Operator Repository Service provides an interface to perform the actions mentioned. It interacts with the Operator Repository Client which resides in each Operator Execution Service instance. The Operator Repository Service is bipartitely organized, which means that there is a secured area to store operators which are only accessible with the corresponding credentials. Each time an Operator Repository Client instance requests an operator, the Operator Repository Service checks if the operator is available without restrictions or whether it needs authentication. In the former case, the operator is accessed and the requested operator package binary is then transmitted to the inquiring Operator Repository Client instance (resident in each Operator Execution Service instance). In the latter case, the inquirer's credentials are validated and the operator package binary is returned if access is granted, otherwise an error message is returned. There might also be other clients interacting with the Operator Repository Service, such as the *NexusDSEditor* presented in Chapter 7.

To uniquely identify an operator, an ID is assigned to every operator inserted in the Operator Repository Service. In order to perform delete or update actions on existing operators, for security reasons the respective operator ID must be known and provided for each of these actions. However, this ID is usually only known to the operator developer and the NexusDS administrator. Operator binaries (see Section 4.2 for details) consist of the meta data part describing the operator, third-party library dependencies which the operator needs to work properly, and the actual operator binaries representing the so-called physical operator. The operator meta data is the relevant part for the Operator Repository Service. The meta data is modeled as an

AWML document. To get operator package binaries the Operator Repository Service must be queried by sending a valid operator ID. This functionality is used by the Operator Execution Service instances in order to retrieve missing operators. The second possibility to query the Operator Repository Service is to send an AWQL query. The AWQL query must contain a predicate which specifies the desired operator type. This query is evaluated and results in a list of operator IDs the query applies to. This functionality may be used by the Core Graph Service to retrieve a list of physical operators related to a logical operator.

Analogous to the Operator Repository Service a storage area for services exist to insert, update, delete and query services. As with operators, each service gets a unique ID when inserted. The Service Repository Service organizes the service package binaries similarly to the Operator Repository Service. The Service Repository Service also features a secure and a public storage area where the services can be stored. Services resident in the secure storage area are only accessible if the credentials provided are valid. The services are packaged in the same way as operators are. They consist of a meta data part, a third-party dependency part, and finally the actual service binaries. As with the Operator Repository Service, the meta data part of service packages is also the most important one for the Service Repository Service. The services are identifiable on the basis of the meta data supplied. An important aspect is, that for delete and update actions it is mandatory to provide the unique service ID, which for security reasons is only known to the service developer and the NexusDS administrator. For query actions either the service ID or the accepted data format type can be provided. The accepted data format type of a service also uniquely identifies a service. Analogous to the Operator Repository Service, if the service is not subject to restrictions, the service package binaries are delivered. Such a restriction might limit service execution to certain nodes. Otherwise the credentials are validated and the service package binaries are only delivered if the inquirer is granted access.

3.6 Compliance of NexusDS with the Requirements

Next, the compliance of NexusDS with the requirements discussed is presented. A brief summary on how NexusDS copes with these requirements is provided, with a reference to the section deepening the respective requirement. The idea behind this is to get a broad overview of the main concepts of NexusDS before going into the details of the single components.

3.6.1 Requirements to the System

I-A. **Custom Data Processing**: Applications formulate their data processing needs by defining a stream processing graph (SP graph) that represents the data sources as well as the data processing sequence. NexusDS offers already many operators which are available for data processing. The NexusDS operator set is extensible, i.e., an application developer can integrate even highly specialized and domain-specific operators to seamlessly

integrate the application with the DSPS. For this, developers of such operators enrich the actual operator implementation by descriptors which are attached to the operator as *meta data describing the operator*. The operator meta data include characteristics such as *accepted and delivered data types*, number of *inputs and outputs*, operator *execution requirements* specifying special software and hardware requirements, or *presets* allowing to specify commonly used settings for the operator parameters. When operators are interconnected, only inputs and outputs of the same accepted and delivered data types can be combined. By building on this flexible meta data concept arbitrary operators can be integrated into NexusDS. For the implementation part many cumbersome tasks and problems are hidden from the operator developer which facilitates the operator development and furthermore reduces development overhead due to redundant components which can be reused in NexusDS.

The details of the operator concept as well as the way NexusDS can be extended with additional operators are given in Section 4.2.

I-B. **Integration of Custom Services**: One way to extend NexusDS—as already discussed—is to integrate operators to express custom data processing functionality. Operators work according to the push-based paradigm and are designed to process data streams in an efficient way. However, besides the push-based functionality we need a mechanism to allow applications to interact with the system on a pull-based or request-response paradigm. This is what services are for: Providing a way for applications to interact with the system and providing non-operational extensibility. Such a service could be a query service for particular sensors providing a dedicated and tailored query language—a so-called domain specific language (DSL)—to query sensor information. Alternatively, we can imagine a domain-specific visualization service for complex rendering techniques which provides a DSL pruning non-needed functionality and tailoring the formulation of processing definitions to the most necessary components. A final example for an application-specific service is a service which accepts SP graphs and, depending on the actual context the application is currently situated, augments the SP graphs by constraints relevant to the application.

The details to this requirement are described in Section 4.3.

I-C. **Dealing with Heterogeneous System Topology**: In order to support a broad variety of potential applications it is important for a DSPS to be open w.r.t. the environments executing the operators. This means that operators may require specialized hardware such as a GPU to perform their task, as demonstrated by the distributed visualization pipeline scenario from Section 2.3.1. To find suitable processing nodes for a specific operator, thus matching operators to processing node, NexusDS operators are annotated with constraints describing the requirements in terms of hardware and software resources [43]. The information about the requirements is used during deployment to constrain the selection of suitable nodes for the specific operator and to guarantee a valid deployment decision. Consequently, the execution environments must be annotated with the same

kind of constraints. This information is used to match operators to concrete execution environments satisfying the operator requirements, thus making their execution possible.

The particulars of this requirement are discussed in multiple sections since this requirement touches many processes and components in the NexusDS system. In Section 3.5.4, the annotation of the execution environments executing the operators has been presented in detail. In Section 4.2.1, the annotation of the operators will be presented and finally in Section 6.5.3 the details on the selection process when distributing the operators is shown.

3.6.2 Requirements to Data Processing

II-A. **Structured and Unstructured Data Support**: NexusDS uses the AWM [105] as the basic structured context-data format. The AWM is an object-oriented, extensible data model tailored to the needs of location-based applications. Like common object-oriented data models, the AWM supports (multi-) inheritance. In contrast to those, AWM objects have no fixed structure but consist of sets of attributes, and the concrete type of the object constitutes an additional attribute. An AWM object can even contain multiple instances of the same attribute, in which additional meta data can be used to distinguish the instances.

For the Nexus system, this concept has two main advantages. Firstly, it greatly facilitates the integration of data coming from different providers. Different representations of the same object can be integrated by unifying the two sets, which even works when the two data providers disagree about the type of the objects. Resolving such inconsistencies can either be done by the system in an additional step, or can be left to the application. Secondly, the concept of multi-attributes allows the representation of dynamic attributes such as the position of a mobile object. In this case, the object contains multiple instances of the position attribute, where each instance contains an additional meta data item representing the temporal validity of this instance.

In addition to AWM objects, NexusDS can also handle application-specific data streams, which allows operators generating, e. g., a video stream (classified as unstructured data). For this, application developers have to implement the specific operators processing the application-specific data. Developers have then to set the corresponding flags in the related meta-data and provide requirements which must be met in order to execute the operator. Developers must also provide the respective serialization and deserialization operators to support distributed processing of their data formats.

Details about the structured data format utilized in NexusDS can be found in Section 2.5.2.1 and in their respective publications [105], [101]. The details about how format specifications are published are presented in Section 4.2.1.

II-B. **Deployment and Execution Specifications**: Data processing in NexusDS is formulated as an SP graph. The Nexus Plan Graph Model (NPGM) and Nexus Execution Graph

Model (NEGM) represent the SP graph format of NexusDS and arrange the operators used for data processing. They support the definition of deployment and runtime constraints. The difference between NPGM and NEGM is that NEGM specifies the whole deployment (physical operators, execution environments etc.) whereas NPGM constitutes a *hybrid graph model* to orchestrate data-flow graphs composed of boxes. Boxes are an abstraction and can either be sources, sinks, or operators. Hybrid graph model means NPGM allows to define properties of the SP graph by *deployment constraints* as well as *runtime constraints*. The annotation of the SP graph by constraints allows influencing the actual deployment process and furthermore defines the runtime behavior of the boxes. By this the concrete implementation of a box or an execution node that is going to execute a box can be defined. NPGM SP graphs are not directly deployable as there may exist boxes that are not mapped to a concrete physical operator (logical boxes) and the distribution, i.e. deployment, of the physical operators is still unknown. Thus, before execution NPGM SP graphs must be mapped to an executable representation (represented by NEGM SP graphs) which in the next step can be deployed and executed on the available infrastructure.

To create an NEGM SP graph the NPGM SP graph is fragmented into subgraphs according to annotated constraints. These fragments are deployed and executed on different heterogeneous and distributed nodes. SP graph fragmentation is a highly complex task. We adopt a meta-heuristic approach that allows us to efficiently find a suitable SP graph fragmentation. By deploying and executing the fragments with their respective boxes on different computing nodes, NexusDS can efficiently process complex tasks such as the streamline calculation scenario (see Section 2.3.1).

Details about the constraint model are described in Section 4.1. In Section 4.5 the SP graph model is discussed in detail. Chapter 6 is entirely dedicated to the mapping and deployment process necessary before execution can start.

II-C. **Exploiting Mobile Devices as Data Source and Execution Nodes**: Nowadays mobile devices have multiple sensors that collect data from the mobile device's context. As shown in the example scenarios in Section 2.3, this data is often important in order to make a stream SP graph work properly, e. g. for setting the area of interest according to the current mobile device's position. Processing capabilities of modern mobile devices have increased in the past decade but are still not suited for execution of complex operators such as the rendering of complex sceneries. In NexusDS, mobile devices can be integrated as data sources as well as processing nodes executing certain tasks filtering data elements before sending them to subsequent processing nodes to reduce bandwidth utilization.

3.6.3 Requirements to Security

For the correct assignment of conditions a reliable authentication of subjects and objects is necessary, i. e. all subjects or objects must be uniquely identifiable and must have the rights to

join the system. The *subject* is an entity which initiates or performs actions in an application. This can be a user, a process, or a service. Subjects address *objects*, which usually are data, such as a stream of images from a camera or a list of subjects. Each subject and object that is to participate in the secure environment of NexusDS must be assigned a unique identity. NexusDS confirms the subject's identity to ensure that this subject is allowed to perform a certain action. This supports liability of action which assigns each action towards an object to a specific subject. To make these actions traceable, a storage area to save the trace information must be provided.

III-A. **Access Control**: Secured objects (objects within the secure area of NexusDS) may be accessed or modified only with proper authorization. This fact presupposes that all subjects and objects operating in the secure area of NexusDS are associated with Access Control (AC) policies. Each time a subject requests to access or modify objects, the admissibility of this access action must be asserted. An access action might be the creation and assignment of new access condition policies for objects. Access actions are only permitted if all subjects and objects involved can be properly authenticated and the permissions apply.

In NexusDS AC policies are defined by data providers making contextual data available within the Nexus system. The corresponding AC policy is defined by associating a certain object (data) to subjects (users) which are allowed access. This means that access to objects might be restricted to subjects resident within a certain domain, e. g. to a set of computing nodes. These policies are stored in the Access Control Service (ACS), which manages the AC policies as well as the Process Control (PC) policies and the Granularity Control (GC) policies. In order to enforce the AC policies, the original SP graph is augmented with associated AC policies and verified before execution. The NexusDS system checks if the AC policies are met before SP graph execution. At runtime NexusDS monitors changes to AC policies and propagates necessary actions to the respective components.

The AC mechanism and the way AC policies are defined and checked are detailed in Section 5.4. The main purpose of AC policies is that objects are protected from unauthorized disclosure and can be provided in different granularity as formulated by requirement III-C, if a corresponding GC policy is defined for this subject-object pairing. This applies even when they are processed by custom-developed operators, preserving the flexibility and openness of NexusDS.

III-B. **Process Control**: Besides AC policies also Process Control (PC) policies exist. In contrast to AC policies, PC policies do not restrict access to objects but rather define how objects can be processed by subjects. This means that the user must have valid PC policies to execute all operators contained in the SP graph. Besides this, operators might have registered PC policies which must be considered, e. g. allowing the execution of an operator only on a specific set of computation nodes. PC policies depend on existing AC policies in the way that there might be a PC policy allowing the execution of a certain operator for a certain user and (due to access limitations) AC policies limit the computation nodes

this operator might be executed on. This means that the subject under concern must also have the necessary permission to access objects.

In NexusDS PC policies are stored in the ACS. The enforcement of PC policies works analogously to the AC policy enforcement. As with AC policies the original SP graph is augmented with associated PC policies and verified before execution. The NexusDS system hereby checks if the PC policies and eventually relevant AC policies are met before SP graph execution. At runtime NexusDS monitors changes to PC policies and propagates necessary actions to the respective components.

The PC mechanism and the way PC policies are defined and checked are detailed in Section 5.4. The main purpose of process conditions is that only subjects having the necessary permission are allowed to process objects.

III-C. **Granularity Control**: As already indicated in the previous sub section, objects should be provided in different granularities to allow fine-grained Granularity Control (GC) policies. By GC policies the fine-grained processing of sensitive objects is possible without threatening privacy issues. This is done by distorting sensible objects and removing information that should not be accessed. The application scenarios discussed in Section 2.3 have higher requirements than simply blocking objects if no access is allowed. Therefore—depending on individual conditions and usage scenarios—in NexusDS a filtering and concealment mechanism of objects is provided which enables to process sensible data by means of multiple LODs. E. g., the LODs of location information may vary depending on the recipient of the information, depending on the question if a recipient is defined as a friend or a work colleague getting either the exact location or only an obfuscated one providing just the city name. As NexusDS makes no restrictions on how information should look like, the security concept included provides a transformation mechanism consisting of a *filter* and an *evaluation* component which is customizable in order to support any transformation and thus any LOD needed.

The definition of fine-grained GC policies is described in more detail in Section 5.4. As this requirement also influences the operator framework provided by NexusDS it is also reflected in Section 4.2.

3.7 Summary

Although many DSPS exist, every one with their own characteristics and their own domain of application, an adaptation problem still persists for the domain of context-aware applications. These context-aware applications rely on the push-based paradigm as DSPS do. Thus, existing functionality in DSPS should be exploited. At the same time these applications often require dedicated processing logics. These should be seamlessly integrated into existing DSPS to prevent the formation of isolated applications. In this chapter we have presented the architecture of NexusDS, a flexible DSPS for context-aware applications. NexusDS is tailored to be customizable by custom services and custom operators.

These components are provided by developers (which usually also develop the context-aware application) and were made available by a repository service. The next chapter describes the SP graphs, the operator and service framework. The source data management of static context data originating from the (former) Nexus system is also described in detail as Nexus allows having multiple data records for the same data object distributed over a couple of data providers. Therefore, special treatment is necessary when retrieving data from the Nexus system.

4 Processing Issues

After presenting the architecture of NexusDS, the processing issues are highlighted in this chapter. In Section 4.1, the constraint model for NexusDS is presented. These constraints naturally arise in the domain of context-aware applications as these applications heavily depend on custom functionality in terms of data processing techniques as well as interaction mechanisms. Section 4.2 and 4.3 describe the operator model and the service model respectively. Each of these components encapsulates a certain functionality of the system and permit a certain processing paradigm being push-based for the operator model and pull-based for the service model. The resource groups clustering NexusDS nodes are introduced in Section 4.4. Afterwards, the flexible stream processing graph (SP graph) model is presented in Section 4.5. The basic structure of a SP graph is a graph interconnecting operators, thus forming a network of operators. In the context of this thesis two different SP graph models exist. These models are called Nexus Plan Graph Model (NPGM) and Nexus Execution Graph Model (NEGM). A NPGM graph provides a logical definition and is not deployable. In contrast to this, a NEGM graph contains all necessary deployment information and can thus be deployed. In DB terminology, the former can be defined as a logical operator graph and the latter one as a physical operator graph. Section 4.6 presents how the requirement constraints that the SP graph formulates are matched against the capability constraints of the available resources in the system. In Section 4.7, the source data management of data originating from Nexus is presented. The source data management concept allows to efficiently retrieve static context data. This chapter is concluded by a short summary in Section 4.8.

4.1 Support for Context-aware Applications

Context-aware applications are demanding and necessitate dedicated mechanisms to integrate domain-specific functionality. To prevent the applications consisting of various isolated solutions and to increase efficiency, applications should be integrated seamlessly. NexusDS

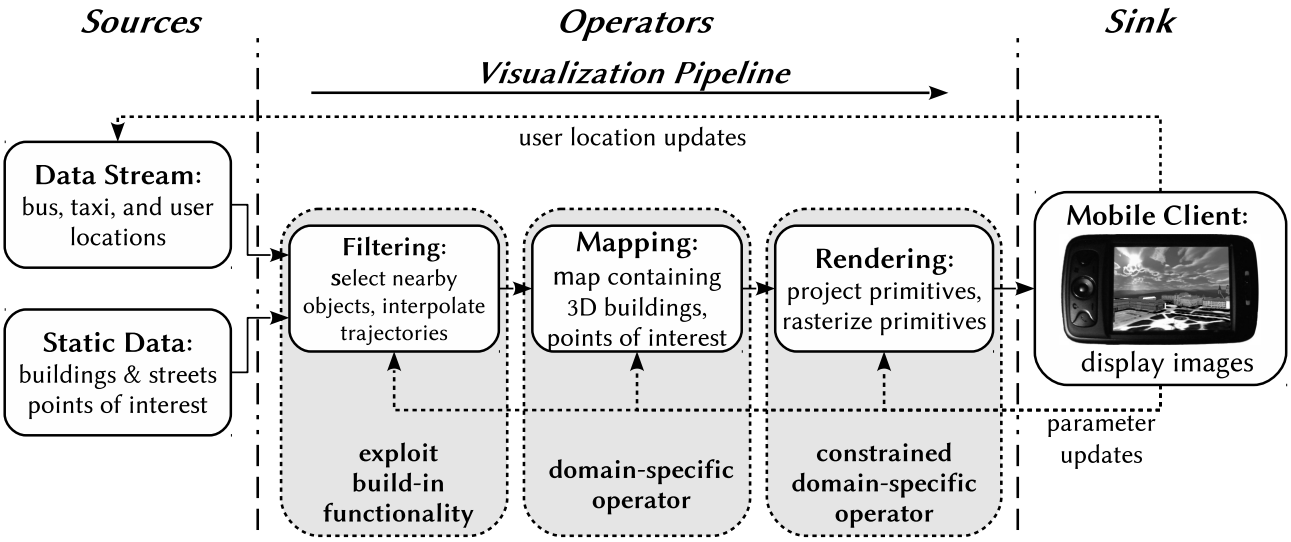


Figure 4.1: Simplified Operator Graph of an Interactive and Location-Aware Visualization Pipeline for Mobile Client Devices.

provides several mechanisms to support this kind of applications. First the constraint model that naturally arises for such systems is presented. Afterwards, we briefly present how the various constraints are taken into account in NexusDS.

4.1.1 Classification of Constraint Types

The constraint model helps to define a classification space for constraints relevant for context-aware applications. This constraint model serves as a base for determining the methods needed for a seamless and adequate integration of context-aware applications.

The context-aware application scenarios presented in Section 2.3 showed that different constraints at different levels of data processing arise. As an example we refer to the *Rendering* operator in Figure 4.1, representing a *constrained domain-specific* operator. This operator is typically meant to run on a GPU. The main reason for this is that a GPU allows to perform the rasterization process efficiently because of the Single Instruction Multiple Data (SIMD) processing technique. There are also other approaches which map originally Central Processing Unit (CPU)-based algorithms to a GPUs [61, 107]. We may think of CPU-based algorithms mapped to FPGA-based algorithms, as proposed by Teubner and Woods [134].

In short, it can be stated that context-aware applications often rely on highly specialized data processing operators. The execution of such operators might be restricted to specialized environments (e. g. for the *Rendering* operator from Figure 4.1). To support these restrictions for highly specialized operators, the system must support the notion of constraints. Constraints are important for the correct deployment and execution of non-trivial SP graphs. This means however, that the deployment algorithm must support constraints to find an optimal set of NexusDS nodes, capable of running the operators of the SP graph. A solution to this problem is presented in Chapter 6.

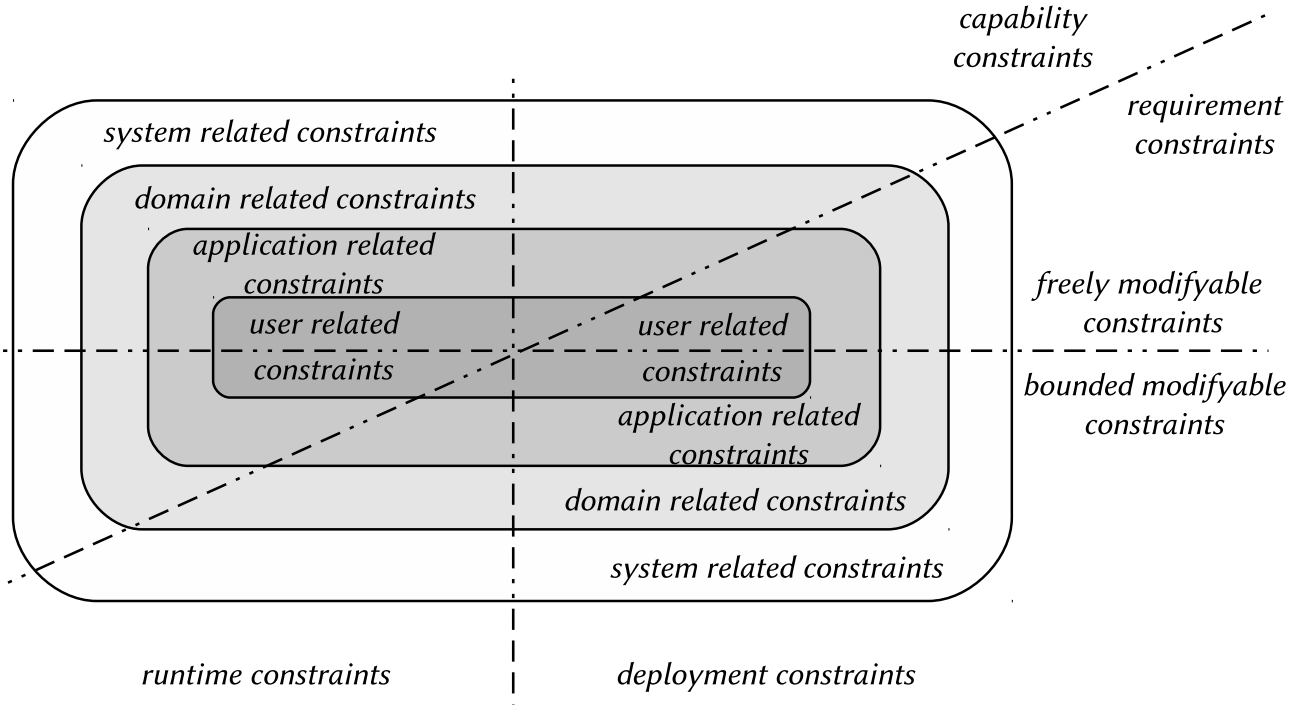


Figure 4.2: The Constraint Space in Data Stream Processing Systems

As shown in Figure 4.2, the resulting constraint space can be subdivided horizontally, vertically and diagonally. Hereby, the constraint space denotes the set of possible constraints. In diagonal direction we define *capability constraints* and *requirement constraints*. In horizontal direction we differentiate between *deployment constraints* and *runtime constraints*. In vertical direction we identify *freely modifiable constraints* and *bounded modifiable constraints*. The resulting partitions are explained in more detail as follows:

Requirement constraints These constraints originate from users, applications, domain extensions and the core system itself. These constraints define the requested features and must be matched against capability constraints. Requirement constraints are integrated as SP graph annotations and this is presented in Section 4.1.2.

Capability constraints These constraints delimit the range of possible requirement constraints configurations, and thus represent the counter part to the requirement constraints. Capability constraints are needed for correct architectural integration and constraint evaluation as described in Section 4.1.2.

It is important to note that in contrast to requirement constraints no user-related capability constraints exist. Users formulate requirements rather than capabilities.

The constraints described above can be overlaid by the vertical classification criteria *deployment constraints* and *runtime constraints*.

Deployment constraints This constraint type influences the deployment process of the underlying system. This in turn reduces the potential search space for the deployment

algorithm searching for suitable NexusDS nodes. As an example, a deployment constraint could state that only Operator Execution Services (OESs) being executed on certain NexusDS nodes and providing a certificate are permitted to participate in processing the data.

Runtime constraints Such constraints influence the runtime behavior of operators and thus influence their execution. This class of constraints has a direct connection to the actual resource consumption of the operator. E. g., a *Rendering* operator can be parameterized in several ways, so that the target resolution is low if resources become lean, and high if there is enough capacity to process it accordingly. This fact in turn has a direct influence on the rendering performance as well as on the actual resource consumption.

Deployment constraints and *runtime constraints* represent the vertical classification criteria for the set of possible constraint types. They can be overlaid by the horizontal classification criteria (*freely modifiable constraints* and *bounded modifiable constraints*), creating six basic constraint space partitions in total.

Freely modifiable constraints In theory *freely modifiable constraints* have no restriction regarding their settings. However, there are restrictions. These restrictions arise depending on the underlying system, e. g., by the hardware engaged and software solutions. Thus, these constraints depend on the design of the system and originate from the system and application developers. These entities define the constraints. E. g. considering the *Rendering* operator, if the resolution parameter (being a *runtime constraint*) is defined as being *freely modifiable*, it is (theoretically) possible to define an arbitrary resolution—obviously as long as the underlying components, i. e. the GPU, support this. For *deployment constraints*, e. g., arbitrary NexusDS nodes can be selected for deployment and execution of this operator as its execution is not restricted to a certain set of NexusDS nodes.

Bounded modifiable constraints These constraints are characterized by the fact that they can only be modified within predefined borders (which are not part of the restrictions that a certain software and hardware combination enforces, as illustrated earlier). This means that the values must be checked against the predefined set of allowed values, defined by the operator developer. In case of the *Rendering* operator, for the *runtime constraints*, a list of allowed values for this parameter is provided, if the resolution parameter is *bounded modifiable*. In this case we cannot select an arbitrary resolution but have to pick one of the values contained in the list of possible values instead. For *deployment constraints*, e. g. a set of allowed NexusDS nodes that should execute the operator can be provided.

These partitions can further be subdivided by the different scopes that occur in DSPSs. As depicted in Figure 4.2, four basic scopes are identified within such systems, namely *system*, *domain*, *application* and *user*. Each scope defines its specific constraints. However, they are not independent from each other. Going from fine to coarse, the dependencies are: *user* scopes depend on *application* scopes, *application* scopes depend on *domain* scopes, *domain* scopes depend on *system* scopes. Thus, the constraint space *user* represents a subset of the constraint

space *application*, the constraint space *application* represents a subset of the constraint space *domain*, and finally the constraint space *domain* represents a subset of the constraint space *system*. In the following, each scope is explained in more detail:

System related constraints Such constraints are defined by the system environment and define the superset for possible constraints (see Figure 4.2). Therefore, the system specific dimensions must be determined and integrated. *System relevant constraints* provide a frame for *domain*, *application* and *user constraints*. E. g., a NexusDS node can be defined having an installed GPU (static information) and a total amount of 2048 megabyte (MB) of Random Access Memory (RAM) (dynamic information). The dynamic information must be observed by monitoring components, ensuring that a certain operator requiring a defined amount of RAM can be successfully executed on a certain node.

Domain related constraints The constraint space defined by the *system relevant constraints* is restricted by *domain relevant constraints*. This allows a better adaptation of the system to specific domain requirements. As depicted in Figure 4.2, the *domain relevant constraints* are completely covered by the *system relevant constraints*. This means no constraints defined by the *system relevant constraints* can be overridden by *domain relevant constraints*. E. g., a developer can define an operator requiring a GPU to run effectively or define an operator to require at least 256 MB of RAM.

Application related constraints Constraints may also be defined for specific applications. These *application relevant constraints* are a subset of the *domain relevant constraints* and allow an application to further restrict the constraint space for a specific operator. E. g., the application may constrain the execution of specific operators to a fixed set of NexusDS nodes. This might be needed to ensure low latencies when interacting with the application by rotating or translating the rendered scene.

User related constraints Finally, *user relevant constraints* represent constraints defined by the user. They are completely surrounded by *application relevant constraints*. These constraints represent a specific user preference. E. g., the user may prefer the scene rendered having a reduced resolution and more details (assuming the operator provides these setting capabilities). Additionally, the user may demand only trusted NexusDS nodes for the execution of the entire data processing. This may be a NexusDS node providing a specific encryption functionality.

Thus, the constraint space defines a frame for the system components and ensures its correct functioning. In the next section the integration of the constraints into NexusDS is presented.

4.1.2 Architectural Integration of the Constraint Model

Figure 4.3 shows the architectural integration of constraints in NexusDS. On the right side *requirement constraints* are shown, which on the other side must be supported by the corresponding *capability constraints* a DSPS offers.

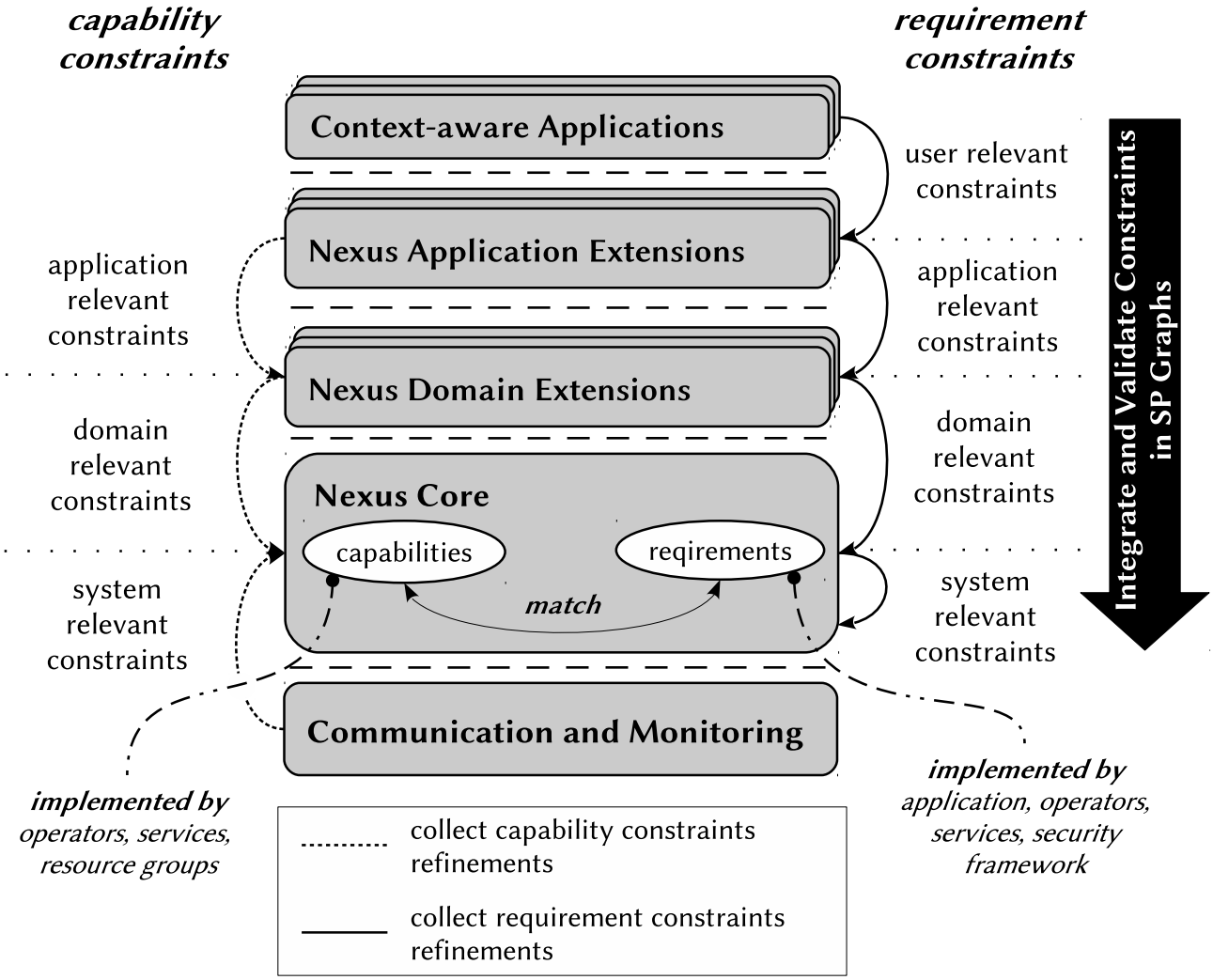


Figure 4.3: Architectural Integration of Constraints in NexusDS

The data structure for propagating *requirement constraints* is represented by *constraint-based SP graphs*. Each layer augments the original SP graph by adding their specific requirement constraints for deployment and runtime. These constraints originate from applications, operator-specific meta data, services and security policies. The different concepts implementing the requirement constraints are context-aware applications, operators, services, and the security framework. All requirements are collected along the path from the top layer to the core layer.

On the other side, *capability constraints* are the counter part to requirement constraints and delimit the possible requirement constraint configurations. Capability constraints are realized by operators, services, and resource groups, defining the available resources for operator deployment and runtime. The capabilities are collected in the core layer. Then, the requirements are matched with the capabilities to validate if the SP graph represents a valid configuration for the system.

For the requirements constraints, going top down, each NexusDS layer adds its specific requirement constraints to the SP graph. The SP graph represents the central structure to transport these constraints to the lower layers. The *user relevant constraints* constitute options which the corresponding context-aware application provides and are defined on the *Context-aware Applications* layer. This layer augments the SP graph by its specific requirement constraints. The augmented SP graph is then passed on to the next layer, which adds its specific requirement constraints. These requirement constraints originate from dedicated services or operators from both *Nexus Application Extensions* and *Nexus Domain Extensions*. Both layers add their *application relevant constraints* and *domain relevant constraints* respectively. The developer formulates the operator specific and service specific requirement constraints. The next and final layer augmenting the SP graph by requirement constraints is the *Nexus Core* layer. In this layer, the *system relevant constraints* resulting from existing core operators are added to the SP graph. Additionally, in this layer the SP graph is augmented by security policies. Therefore, all relevant security policies are extracted and added to the SP graph. The security framework and its related concepts are presented in detail in Chapter 5.

Now, the SP graph has been augmented by all relevant requirement constraints, symbolized by the *requirements* in the core layer. These requirement constraints must be validated in the next step. Thereby, the requirements should obviously not violate the *capabilities*. The capability constraints are also collected by the respective entities. The *Communication and Monitoring* layer build the basic capabilities for deployment and runtime of operators. Besides, the *Nexus Application Extensions* and *Nexus Domain Extensions* provide capability constraints by, e. g. limiting the usable amount of main memory per operator. Furthermore, operators themselves provide capability constraints defined by their respective developers. These capability constraints include the anatomy of the operator itself, i. e. its parameters. Additionally, a developer can limit the range of possible values for the operator's parameter.

Only SP graphs whose requirement constraints match the capability constraints available, i. e. whose requirements are satisfied, can be deployed and executed. Thus, before SP graph deployment and execution, the requirement constraints must be matched against capability constraints a DSPS offers. These constraints naturally limit the possibilities for deployment and execution.

The requirement and capability constraints are implemented by different concepts as shown in Figure 4.3. In the next sections the concepts for realizing the constraint model are presented in detail. First, in Section 4.2 the operator model formulating requirements and defining the constraint space for the upper layers are presented. In Section 4.3 the service model is introduced, allowing to integrate highly specialized functionality to manipulate SP graph structures. The resource groups are presented in Section 4.4. Finally, the concept of a constraint-aware SP graph and the matching concept of requirement constraints to capability constraints are presented in Section 4.5 and Section 4.6. The security framework which is related to the constraint concept is presented in its dedicated Chapter 5.

4.2 Operator Model

Generally speaking, operators implement a push-based principle. Thereby data streams originating from a source or other operators are received by their associated inputs and processed according to the implemented processing logic. Although the Augmented World Model (AWM) represents the main data model in NexusDS, the operator model also allows to create operators processing arbitrary data. Operators for context-aware applications range from basic operators such as *selection* or *union* to highly domain-specific operators implementing e. g. the *rendering* of a complex scene (within a visualization pipeline) or image processing, such as *face recognition* for a stream of images.

The operator model of NexusDS differs from others, as the ones proposed in [2, 9, 127]. In contrast to these approaches, our operator model approach is built on a meta data description and allows to implement and integrate customized operators for a specific domain. The operator-specific meta data describes the operator's characteristics and enables the system to group operators according to their logical operation, and perform plausibility checks if the operator is executable on a certain NexusDS node. The meta data contains information, e. g. on the *accepted and delivered data types*, the number of *inputs and outputs*, the operator *execution requirements* specifying special software and hardware requirements, or *presets* specifying the default setting of operator parameters as well as commonly used settings for the operator parameters. The operator meta data is stored in the Operator Repository Service and is exploited by the Core Graph Service to find suitable mappings from operators to computing nodes.

A detailed view on the operator model is depicted in Figure 4.4. The operator model is subdivided into three parts: *Input queues* handling incoming data streams, the *input manager* synchronizing the single inputs, and an *operator* actually processing the incoming data. These three components are encapsulated in a *box*. A box is a generic container that allows to embed the proper operator logic easily and reuse already existing components. Thus it is possible to combine new operators with already existing queues, handling input data streams in a certain manner and input managers implementing a certain synchronization scheme without the need to re-implement all components needed from scratch for each operator.

The operator model distinguishes between two component types w.r.t. their data retrieval behavior: *Passive* components and *active* components. Passive components are data-driven as they do not actively pull data but instead are triggered as soon as data is present on the input. On the input side they wait for data elements to arrive before they start processing them. On the output side, the data is pushed to subsequent components. Contrarily, active components are not data-driven but process-driven in the sense that they are not triggered by data. On the input side these components thus implement a pull-based approach to retrieve data. On the output side these components then push the data to subsequent components. The only active element in the operator model of NexusDS is the input manager. It actively pulls data from the queues and furthermore synchronizes the inputs of the operator according to a predefined synchronization scheme. The queues and the operator are passive elements in the boxed operator model and are triggered on data reception. The decoupling of the synchronization logic

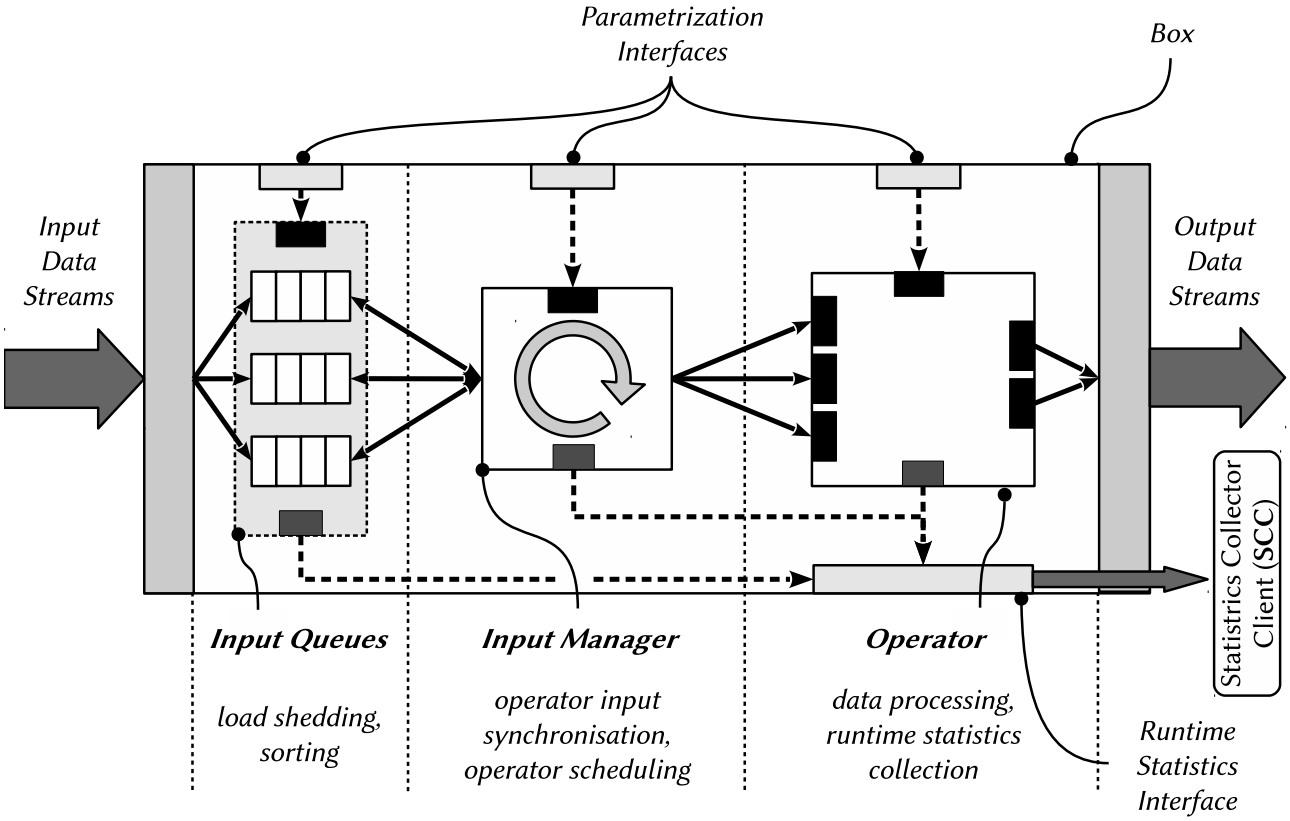


Figure 4.4: The Boxed Operator Model in NexusDS

from the operator and the fact that the operator is a passive element alleviates the implementation efforts of the actual operator logic. I. e., operator developers can rely on already existing synchronization schemes, as the input manager provides the necessary data for one operation cycle[1]. This model allows to easily create custom operators, and reduces development efforts as re-usability of already existing components is high. The components of the operator model depicted in Figure 4.4 are described as follows:

- **Box**: A box represents the single distribution unit and contains all components necessary for operator execution. The box is subdivided into three areas, each one containing a different component with its specific functionality. The first area consists of input queues, the second contains the input manager, and the last one the operator. The Box offers a *parametrization interface* to manipulate the parametrization of the single components. Furthermore, a box integrates a *runtime statistics interface* that retrieves statistical information on the runtime behavior and status of the operator, the input manager, and the queues. These statistics are then sent to the Statistics Collector Client resident within the Operator Execution Service. A box receives the input data streams and locally routes them to the associated queues. Also, once the operator has processed the data, its box routes the resulting output data streams to its attached subsequent boxes.

[1]In the context of this thesis an operation cycle represents the application of the processing logic implemented by an operator to the data.

- **Input queues**: An input queue is a synonym for the concept of a window, as used in the context of DSPS. Input queues are necessary because the flux-frequency of incoming data elements might be higher than the processing frequency of the operator. This might occur arbitrarily during the execution of an operator. Without input queues, data elements might be missed, resulting in incomplete data [60]. Based on their individual characteristics, input queues can be classified into different input queue types. The most commonly used ones are *time-based* and *count-based* input queues. The life cycle of the data elements depend on the input queue type's characteristics. For count-based input queues this depends on the actual input queue size, i. e. the maximum number of the stored data elements. The life cycle starts when the data element arrives at the input queue, and ends when it is expunged to make room for new ones. For time-based input queues, the life cycle of data elements is given by the maximum time period the input queue holds a data element. Each data element has a corresponding time stamp. In this case the life cycle for a data element starts when it arrives at the input queue and ends when the current time period expires (e. g. by a sliding window semantic). Furthermore, input queues can be exploited to sort incoming data according to a local sorting policy, e. g. sorting them by priority or by timestamps. It is important to note that sorting only occurs within the context of the input queue. Otherwise, sorting would have the behavior of a blocking operation [16].

  However, the use of input queues does not prevent overload situations, requiring the application of stream filtering techniques, such as load shedding [131] or aggregation [14]. By integrating this functionality into input queues, the overhead of integrating additional *drop* operators within the already running execution graph is avoided, as proposed by Tatbul and Zdonik [131].
- **Input Manager**: The input manager handles data transmission from input queues to operators. Thus, the input manager influences the synchronization of the operator inputs. Various synchronization schemes can be implemented, i. e. the order in which the input queues are queried for available data. These techniques are implemented once and used many times. The input manager may also have parameters to fine-tune its behavior. Additionally, the input manager is exploited for operator scheduling tasks. Therefore, the input manager receives scheduling orders from the Operator Scheduler, influencing the execution order of the operators.
- **Operator**: Operators process the data streams. Within a box a certain arbitrary operator can be embedded. Like input queues and input managers, operators also have a parameterizable interface to pass operator-specific parameters to influence the operator's processing. E. g. for a visualization operator the resolution at which a scenery is rendered is a possible parameter. For a selection operator the parameter might be the selection predicate to filter data elements. Also, the operators have multiple input slots and output slots, each representing data necessary for the operator to render the scenery. Each slot might accept a different data format, e. g. one input slot accepts data originating from Nexus constituting an AWM result set representing the surrounding buildings. Another

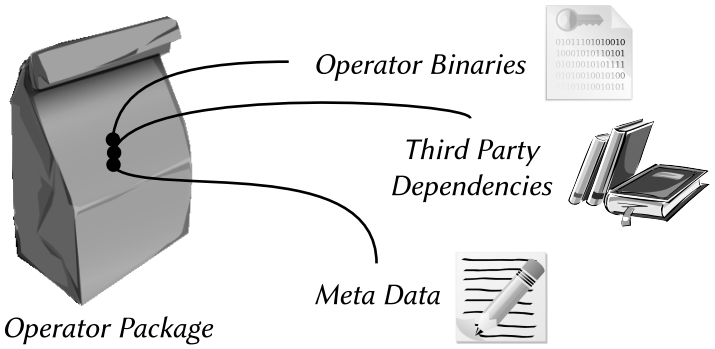


Figure 4.5: Structure of Operator Packages in NexusDS

input slot accepts a proprietary binary format consisting of models for the single object types to be rendered. Analogous to the input slots also output slots might have different data delivery types, e. g. resulting images from the rendering process.

The operator components are stored as packages in the Operator Repository Service. The structure of such a package is depicted in Figure 4.5. The package consists of three components. The first component is the actual operator represented by the *operator binaries*. These binaries are created by extending the operator framework already existing. The second component are possible third-party dependencies that are necessary for the operator to run properly. Usually these dependencies constitute programming libraries that are used by the operator binaries. The third and final component is the operator meta data, describing the special characteristics of this operator. For input queues and input managers the package contains similar components, albeit the particular binaries represent either an input queue or an input manager and its respective meta data, which describe the characteristics of the components. The meta data is detailed in the next section.

4.2.1 Operator Meta Data

As described in [45], the development and provisioning of context data should be done by context models [62]. In this sense, domain-specific extensions of the NexusDS system correspond to the (technical) context for a particular application or domain of interest. In general, context models may contain geographical data (e. g. maps), dynamic sensor data (e. g. the position of a moving object), infrastructure data (e. g. the extent and bandwidth of wireless networks), or context-referenced digital information (e. g. documents or web sites that are relevant in a certain context). Therefore, in a broader sense, context models can contain information on technical context such as available computing nodes or data processing operators and their respective properties. We model the meta data of NexusDS-related components as so-called *technical context* within the AWM, the shared context model of the Nexus system.

The operator-specific meta data announces information on the operator characteristics, thus describing the operator. This concept allows integrating arbitrary operators in NexusDS. Operators in NexusDS may have special characteristics that must be taken into account when

modeling the operators' meta data [43]. From a data modeling point of view, the following requirements must be satisfied:

- The meta data must describe the operator's characteristics to allow the NexusDS system to perform plausibility checks without any knowledge of the real implementation.
- In order to successfully execute an operator, the NexusDS system needs meta data that defines the operator requirements, e. g. the execution environment it needs.
- The meta data must provide preset values for the operator parameters. This basically guarantees correct execution and represents the default parametrization of the related operator.

We have divided the operator meta data in three parts, namely *Operator Descriptor* describing the fundamental structure of the operator, *Operator Requirements* declaring the requirements in terms of software and hardware, and finally *Operator Presets* providing default parameter values for the operator parameters. The following subsections deal with these components.

4.2.1.1 Operator Descriptor

The *Operator Descriptor* contains information on the operator and describes its main characteristics, such as the accepted and delivered data types, the number of inputs and outputs, dependencies to other operators, or the number and types of parameters the operator has. These details are used by NexusDS to check for compatible inter-operator communication patterns in SP graphs. The operator descriptor is modeled as part of the AWM by extending the schema information as described in Section 2.5.2.1. Therefore, we have exploited the tools that we developed to support developers for their NexusDS-related development and have utilized the NexusDSEditor, presented in Chapter 7.

The operator meta data are represented as the technical context within a context model [101, 103, 105]. In order to reflect these component types within the AWM, an Extended Attribute Schema (EAS) and an Extended Class Schema (ECS) have been created. These schemas are written in XML. The base class for all classes of the AWM is the class called **NexusDataObject** and all further classes are derived from it. The **NexusDataObject** class defines basic attributes such as **name** or **description** of its object instances. These attributes are inherited by the **NexusExecutionEnvironmentElement** class which is the basic class for all data processing related components of NexusDS. This class defines basic attributes. These attributes include **author**, identifying the author of a certain execution-related component and **nativeLibrary**, defining if this execution component relies on native libraries that must be loaded by the platform in order to execute components properly.

The class **NexusExecutionEnvironmentElement** is further refined to dedicated classes for each execution component type, i. e. input queues, operators, and so forth. The respective classes define the relevant attributes and enable the NexusDS system to get information on the operator, e. g. for plausibility checks of interconnected inputs and outputs in an NPGM SP

graph (see Section 4.5). In the following, we describe the operator's **slot** attribute in detail, defined by the **NexusSlotAttributeType**) shown in Listing 4.1.

```
1  [ . . . ]
2
3  <!−− Operator Descriptor Attribute Definition − OperatorDescriptorAttributeSchema −−>
4  <complexType name="NexusSlotAttributeType">
5    <annotation>
6      <documentation>A complex Nexus Attribute Type that contains the inputs and outputs of an
           entity</documentation>
7    </annotation>
8    <sequence>
9      <element name="value">
10       <complexType>
11         <sequence>
12           <element name="name" type="nsat:NexusStringType" minOccurs="1"/>
13           <element name="id" type="nsat:NexusIntegerType" minOccurs="1"/>
14           <element name="opClass" type="nsat:NexusStringType" minOccurs="0"/>
15           <element name="type" type="nsat:NexusStringType" minOccurs="1"/>
16           <element name="connectOptional" type="nsat:NexusBooleanType" minOccurs="0"/>
17           <element name="media" type="nsat:NexusStringType" minOccurs="1"/>
18           <element name="mediaTypeAWM" type="nsat:NexusTypeType" minOccurs="0"/>
19           <element name="mediaTypeClassURI" type="nsat:NexusUriType" minOccurs="0"/>
20           <element name="mediaTypeClassURIFactory" type="nsat:NexusUriType" minOccurs="
                 0"/>
21           <element name="mediaAccessType" type="nsat:NexusStringType" minOccurs="0"/>
22           <element name="attributeSchema" type="nsat:NexusStringType" minOccurs="0"/>
23           <element name="classSchema" type="nsat:NexusStringType" minOccurs="0"/>
24         </sequence>
25        </complexType>
26     </element>
27     <element ref="nsas:meta" minOccurs="0"/>
28   </sequence>
29 </complexType>
30
31 <element name="slot" type="eas:NexusSlotAttributeType" substitutionGroup="
       nsas:NexusAttribute"></element>
32
33 [ . . . ]
```

Listing 4.1: Excerpt of the Extended Attribute Schema representing the Operator Slot Attribute.

As shown, the **slot** attribute is represented as a complex attribute type **NexusSlotAttributeType** (lines **4** to **29**). The attribute declaration is shown in line **31**, which is used in the class

schema for class definition. For the rest of this section, an operator is the execution component under concern. For the other execution components the attribute structures are similar and thus not provided. The **slot** attribute consists of the following attribute parts:

name This is a human readable name of the slot. It serves to write status information to log files. It also assists developers building SP graphs, as these names are easier to understand than abstract identifiers. This attribute does not necessarily have to be unique, however it is sensible to do so.

id This attribute is an ID uniquely identifying this slot. It is used to interconnect outputs of the operator with the corresponding inputs of another operator in a NPGM or NEGM SP graph.

type This attribute defines the type of the slot. It can be either of *input*, of *output*, or of *parameter* type.

opClass This attribute groups operators. It is useful if, e. g. a developer provides different implementations for the same logical operator. In this case, in a SP graph a box is being defined as belonging to a certain operator class. In the next step, the query processor, i. e. the Core Graph Service, must map this to a real physical operator contained in the defined group. This attribute part is optional.

connectOptional Defines if the slot must be connected or not. Per default all inputs must be connected to some output of previous operators. This semantics is used since the system assumes that data from each input slot is needed to produce output data. If an input slot is declared optional, it is declared as being not absolutely needed for the related operator to work correctly. In contrast to input typed slots, per default output and parameter types slots must not be obligatorily connected to some input.

media Declares what kind of media is accepted by this slot. It can be either *stream* or *set*. This allows the query processor to perform specific optimizations, e. g. if one input slot is defined as stream and the other as set, then the buffer size of this operator—assuming that there is enough space for doing that—can be set equal to the cardinality of the set to be received, so avoiding incomplete results as they may appear if we miss to evaluate this fact.

mediaAccessType This attribute can be set to either *write* or *read*. If an operator modifies incoming data elements by overwriting them, writing is mandatory. Otherwise, this attribute should be set to read. The system per default assumes the incoming data is accessed only by read operations and the operator produces new output data if necessary. Especially in a complex scenario, where many operators being executed on the same machine receive the same object instances, setting this attribute part to read enormously increases scalability in terms of memory consumption, since only references to objects are passed and the system does not have to create a deep copy of each incoming object before passing it on to the processing instances.

mediaTypeClassURI This attribute defines the class types of the incoming data elements. If this attribute part is set to **java.lang.String**, the data is treated as a Java String object. However, an arbitrary class Uniform Resource Identifier (URI) can be provided here, specifying the data type for this slot.

A special case is represented by the value **nexus.awm.GenericObject**. This indicates that a slot is going to receive (or send, depending on the slot type) AWM data objects. In combination with the **mediaTypeAWM** attribute this can further be refined, i. e. by the concrete AWM class that is accepted or delivered by this particular slot.

mediaTypeClassURIFactory This attribute is optional and points to the factory able to create instances of the **mediaTypeClassURI** object types. It gives developers the opportunity to provide dedicated factory classes which perform necessary initialization and creation tasks.

attributeSchema If the data type this slot deals with is AWM-related data, and the AWM class is not part of the standard schema, this attribute part defines the Extended Attribute Schema (EAS) file needed in order to recognize the attributes the new class type depends on.

classSchema As with the **attributeSchema** attribute part, this attribute part specifies the Extended Class Schema (ECS) file for additional AWM classes.

Besides the **slot** attribute, several other attributes are needed to fully describe the operator. These attributes include:

inputManagerDependency This attribute specifies dependencies to an input manager, thus implicitly defining the synchronization scheme needed for this operator. It provides the reference to the corresponding input manager. If this attribute is not set, the default input manager that is used implements a *Round Robin* synchronization scheme.

queueDependency Like the **inputManagerDependency** attribute, this attribute describes the dependency of a certain slot (either input, output or parameter) to an input queue type. Therefore, this attribute provides the slot ID and the reference to the input queue that must be used with that slot.

preParallel and postParallel These attributes specify the specific *demultiplex* and *multiplex* operators necessary to parallelize the operator the descriptor belongs to. Before deployment, the Core Graph Service transforms the SP graph by placing the operators in front of or behind the respective operator. As many operator instances of the operator to parallelize as needed are then created and interconnected with the multiplexing and demultiplexing operators.

preTransmit and postTransmit These attributes define how the data must be serialized and deserialized when crossing node borders. Therefore, they point to other execution environment components called *platform sinks and sources* for serializing and deserializing the data. The NexusDS system then sets these platform sources and sinks up on the corresponding NexusDS nodes that sends and receives the data.

preDependency and postDependency These attributes define dependencies of an operator to other operators. Hereby, the **preDependency** attribute defines dependencies to an operator that must be placed before this operator. In contrast to this **postDependency** defines operators that must be placed behind this operator. The transformation operations are performed by the CGS.

pinnedTo This attribute is optional and specifies the NexusDS nodes this operator is pinned to. That means the operator developer can specify on which NexusDS nodes the operator can be executed.

4.2.1.2 Operator Requirements

The *Operator Requirements* specify the requirements of an operator in terms of runtime constraints and deployment constraints. These requirements are matched to the available NexusDS nodes and their capabilities. Since NexusDS has been designed for a wide variety of devices, this information is crucial for the correct deployment and execution of the operator. Similar to the operator descriptor, the operator requirements are also modeled as AWM objects. Listing 4.2 shows an excerpt of the EAS definition for the operator requirements attributes.

```
1 [...]
2
3 <!-- Operator Requirements Attribute Definition - OperatorRequirementsAttributeSchema --
      >
4 <element name="key" type="nsas:NexusStringAttributeType" substitutionGroup="
      nsas:NexusAttribute"></element>
5 <element name="value" type="nsas:NexusStringAttributeType" substitutionGroup="
      nsas:NexusAttribute"></element>
6 <element name="requirementTypeClassURI" type="nsas:NexusUriAttributeType"
      substitutionGroup="nsas:NexusAttribute"></element>
7 <element name="requirementTypeClassURIFactory" type="nsas:NexusUriAttributeType"
      substitutionGroup="nsas:NexusAttribute"></element>
8
9 [...]
```

Listing 4.2: Excerpt of the Extended Attribute Schema representing the Operator Requirement Attribute.

The requirements are modeled as a **key–value** combination. Thus, operators and NexusDS nodes are tagged according to their requirements and capabilities, i. e. by tagging the corresponding keywords. This allows an operator-level virtualization. The *key* can be either *software* or *hardware*. The *value* is an arbitrary identifier to uniquely identify the requirement constraint. The value is mapped to the corresponding resource group. It is important to note that the values can be arbitrarily chosen but should correspond to the values of the external-

ized capabilities of NexusDS nodes. Otherwise the two constraint types (requirements and capabilities) cannot be matched.

Beside the key and value attributes, there are also the **requirementTypeClassURI** and **requirementTypeClassURIFactory** attributes. The former attribute declares the Java class type of the value data type to handle them correctly. The latter attribute specifies a factory to create object instances of the provided **requirementTypeClassURI**.

4.2.1.3 Operator Presets

Operator Presets specify commonly used settings for the operator parameters. This is beneficial, especially if an operator has many parameters. The operator developer must provide at least one preset for each parameter. This is necessary to ensure the correct execution of the operator. The presets are also defined by a key–value pair schema, whereby *key* identifies an operator's parameter and *value* specifies the corresponding parameter value to be set.

4.3 Service Model

The service model in NexusDS is similar to the operator model presented in Section 4.2. A service in fact also implements a well-defined interface and is packaged with necessary third-party libraries and meta data describing the service. It describes the service's accepted document formats, its corresponding resource group and so forth. As described in Section 4.4, a resource group limits the scope of its components. This means that only those entities can invoke the service which are in the same resource group.

Services implement a pull-based communication paradigm. This means that a service operates and performs its specific actions each time it is invoked. Thereby, a service is typically invoked by applications. But services might also be invoked by other services, e. g. the Core Graph Service which invokes the Operator Execution Service to deploy the SP graph fragments. NexusDS offers a set of basic services which form the core service functionality of NexusDS. These services have already been described in Section 3.5. Additional services can be implemented and seamlessly integrated into NexusDS providing highly domain-specific service offerings. Such a domain-specific service could be a dedicated visualization service which accepts a query formulated in a domain specific language (DSL). Thereby the specialized DSL offers adequate querying primitives for the respective domain of interest.

A detailed view on the service model is depicted in Figure 4.6. As presented in Section 3.5.1, the Service Manager controls the life cycle of a service. The first step of a services life cycle is to initialize the service (1). For that, the service developer can specify the initialization routine of the service. This is an optional step and might be necessary if the service needs to initialize more complex data structures which are needed during the service startup routine. Then, the service is started and a registration message is sent to the Service Repository Service (2). This makes the service discoverable. After the start routine has finished, the service

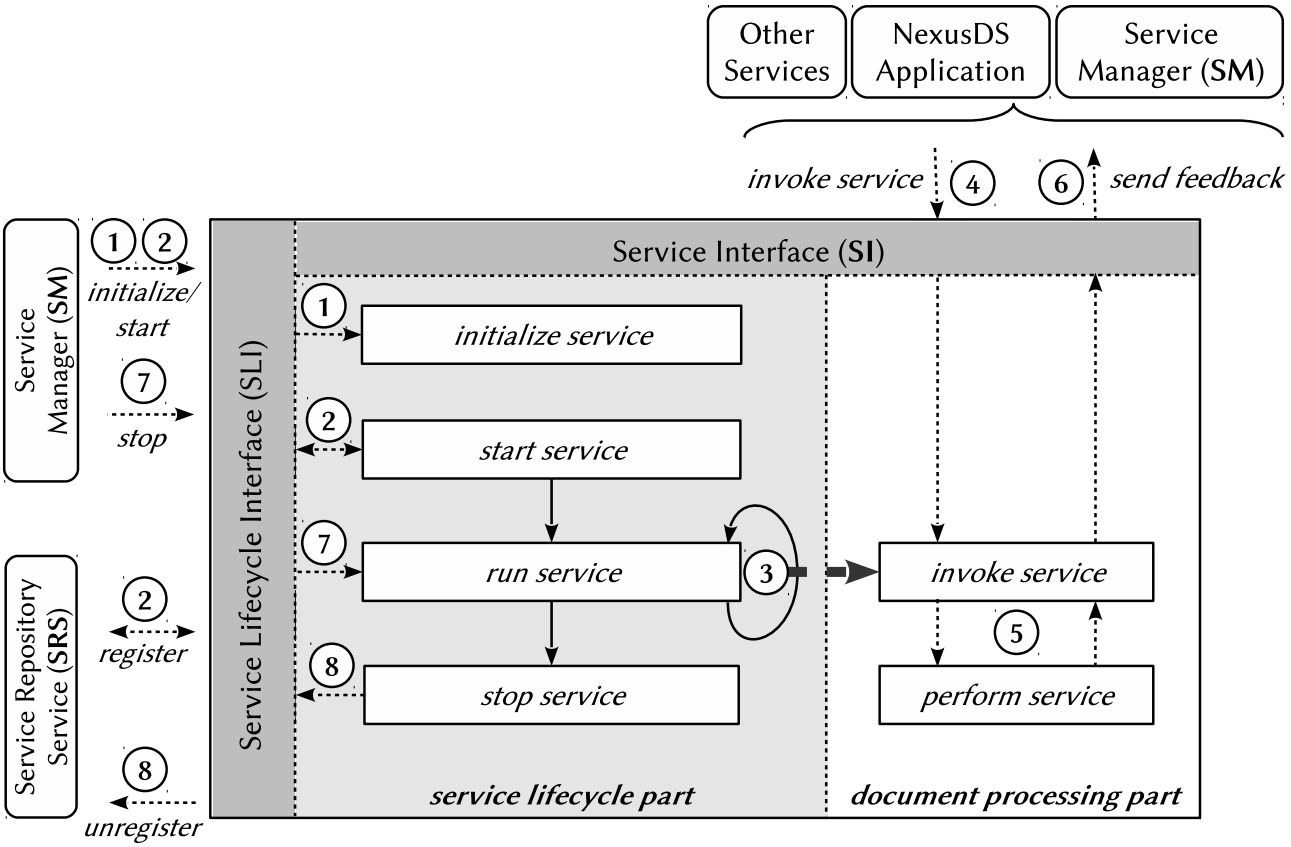


Figure 4.6: The Service Model in NexusDS

offers its invoke functionality (3). From now on, the service can be invoked by other services or applications (4). The document exchange format for service invocation and the resulting service's response document is XML. The invocation document is then passed on to the actual service implementation and the document is processed (5). The service returns the results to the inquirer (6). As long as the service is running the service can be invoked. Once the service is no longer needed, its life time has expired. Thus, the service should be stopped. The Service Manager sends message to stop to the service (7). Once this message is received, the shutdown procedure for this service is initialized (8). Before stopping the service, it is logged off from the Service Repository Service and currently running processes are stopped.

As depicted in Figure 4.6, the service model consists of two parts: The service life cycle part on the left side (light gray shaded) and the document processing part on the right side. For the service life cycle part the initialization routine for the service is optional. The remaining steps of the service life cycle part are fixed and perform the operations in a well-defined order. However, the implementation of the invocation module (on the processing side) is mandatory as it represents the actual service implementation. Basically, an interface represents this module which a developer must implement appropriately.

Besides the implementation, a number of additional meta data is required. This meta data describes the service characteristics and also enables to influence the service execution.

4.3.1 Service Meta Data

As the operator-related meta data, the service-related meta data is also modeled as technical context in the AWM. This concept allows integrating arbitrary services in NexusDS and exploiting the capabilities of the AWM. The service-specific meta data describes the service. From a data modeling point of view, the following requirements must be satisfied:

- The meta data must fully describe the service. This includes its exchange document format and its scope.
- In order to successfully execute a service, the NexusDS system needs the specification of the environment necessary for the correct service execution.

The service meta data consists of two parts: A *Service Descriptor* part which describes the characteristics of the service and a *Service Requirements* part declaring the requirements in terms of software and hardware for the service in order to run properly.

The service requirements meta data are similar to those for operators, so they are not described in detail. Section 4.2.1.2 provides this in more detail. The descriptor differs from the one for operators as other characteristics must be represented. In the following, we exemplary describe the **dependency** attribute depicted in Listing 4.3.

```
1  [ . . . ]
2  <complexType name="NexusDependencyAttributeType">
3    <sequence>
4      <element name="value">
5        <complexType>
6          <sequence>
7            <element name="dependencyURI" type="nsat:NexusUriType" minOccurs="1"/>
8            <element name="dependencyMedia" type="nsat:NexusStringType" minOccurs="1"/>
9            <element name="attributeSchema" type="nsat:NexusStringType" minOccurs="0"/>
10           <element name="classSchema" type="nsat:NexusStringType" minOccurs="0"/>
11         </sequence>
12       </complexType>
13     </element>
14     <element ref="nsas:meta" minOccurs="0"/>
15   </sequence>
16 </complexType>
17
18 <element name="dependency" type="eas:NexusDependencyAttributeType" substitutionGroup="
       nsas:NexusAttribute"></element>
19 [ . . . ]
```

Listing 4.3: Excerpt of the Extended Attribute Schema representing the Service Dependency Attribute.

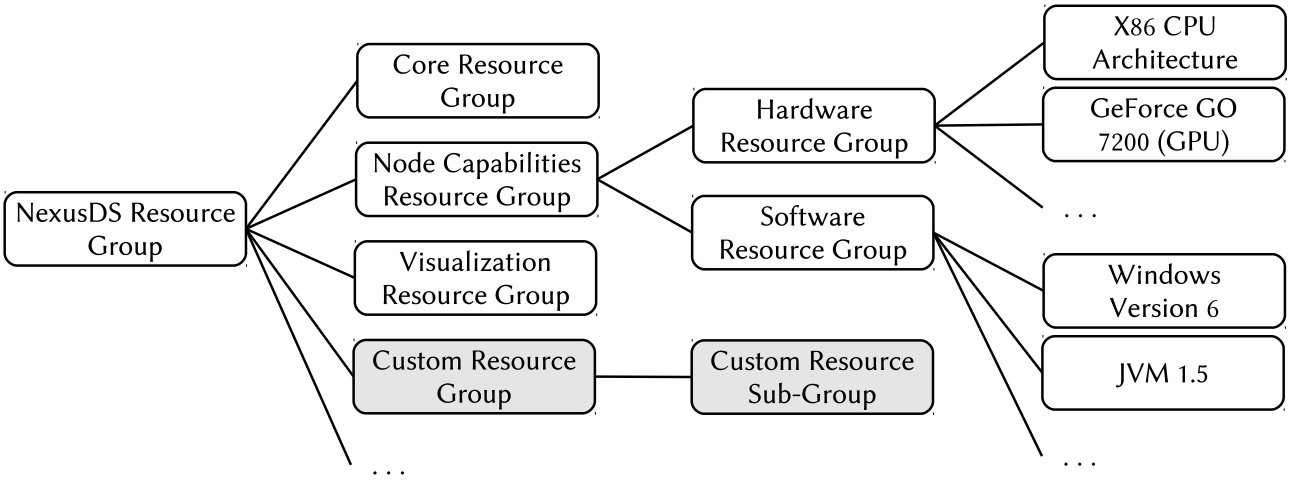


Figure 4.7: Resource Groups in NexusDS

A service description consists of the attributes **author**, **pinnedTo**, and **dependency**. The latter attribute is depicted in Listing 4.3. As shown, the dependency attribute is represented as a complex attribute type (**NexusDependencyAttributeType**) and the attribute declarations (line **20**), which are used in the respective class schema for class definition. Thereby, the **dependency** attribute defines a service B on which a service A the descriptor belongs to relies on. This means, that service A necessitates a service B specified in the **dependency** attribute in order to run properly. In other terms, an instance of service B must be running and available. The remaining attributes are self-explanatory and denote the same as with operator-related meta data, as described in Section 4.2.1.1. These attribute definitions are not shown in the listing.

Furthermore, the descriptor optionally specifies the resource group to which the service belongs to. If there is no such definition, the service is assigned to the default resource group. Resource groups, beside others, limit the scope of services. Resource groups cluster entities that share a common scope. By defining a service belonging to a certain resource group, only entities, i. e. applications and services belonging to this resource group can discover and use this service. Thereby, applications and services may belong to several different resource groups.

4.4 Resource Groups

The *resource groups* concept provides a base for the capability constraints and capture static information on the NexusDS nodes. Resource groups split up the resources into different, not necessarily disjoint, clusters. This means, that in order to get access to a certain service belonging to a resource group, the entity trying to access needs the corresponding permission to join the requested resource group. This means that applications which try to use a service in a certain resource group must first have access to the respective resource group. Thus, the utilization of resources belonging to a certain resource group is restricted to these entities.

Figure 4.7 shows the hierarchical structure of the resource groups. The root is represented by the *NexusDS Resource Group*. Resources belong at least to one resource group and may belong to many resource groups at the same time. All components of NexusDS—all available services and operators as well as applications—per default belong to this group. I. e., if an operator is not defined to belong to a specific resource group it is automatically assigned belonging to the *NexusDS Resource Group*. Besides this root resource group additional resource groups can be defined. NexusDS here defines the *Node Capabilities Resource Group*, grouping all available processing nodes according to their characteristics. Beside this, the *Core Resource Group* exists, grouping all services and operators available within the system. Arbitrary additional resource groups can be defined together with the already existing ones. Resource groups are defined by domain or application extension developers or system administrators. Such a custom resource group is, e. g. the *Visualization Resource Group* depicted in Figure 4.7. This Resource group might group all functionality belonging to the domain of a visualization scenario.

By default, resource groups are public, meaning all services and operators are accessible or executable within their correspondent resource groups. However, this is not always intended as there is also functionality that should not be accessed arbitrarily. Therefore, trusted resource groups can be created, as the *Custom Resource Group* example displayed in Figure 4.7 shows as a gray box. Entities requesting access to the resources contained in such a resource group must first provide valid credentials before they are allowed to access them.

The system-relevant constraints are modeled by the *NexusDS Node Resource Group* and its underlying resource groups. E. g., all NexusDS nodes having a *x86 CPU architecture* are arranged within the corresponding resource group. This is beneficial, as search space for finding suitable NexusDS nodes for the operators might be drastically reduced in size by picking only the resource groups of interest.

4.5 Stream Processing Graph

The SP graph defines the data processing as it consists of different operators that are interconnected, forming a processing pipeline. Furthermore, the SP graph represents the central exchange format for constraint propagation between the single layers. Two different SP graph formats are distinguished: The Nexus Plan Graph Model (NPGM) and the Nexus Execution Graph Model (NEGM). They differentiate from each other as NEGM formatted SP graphs provide full and unique deployment information whereas NPGM formatted SP graphs do not. The main idea is that each layer augments and modifies the original SP graph by adding their respective requirement constraints. The constraint annotations originate from user and application preferences, operator-related requirements for deployment and execution, or domain-specific services. Constraint annotations represent a universal mechanism to integrate highly domain-specific knowledge and thus influence the deployment as well as the execution of SP graphs. First, the NPGM and afterwards the NEGM SP graphs are presented in more detail.

4.5.1 Nexus Plan Graph Model and Nexus Execution Graph Model

The NPGM is a flexible composition model to orchestrate data-flow graphs. As depicted in Figure 4.8, an NPGM formatted SP graph consists of a set of interconnected boxes that constitute either a source operator, a sink operator, or an operator. NPGM boxes have an arbitrary (but well-defined) number of connection slots. Each connection slot is uniquely identified and can be either an input or output. The box-specific implementations describe the expected and delivered types respectively. Differences in data types are denoted by the different shades of gray used for the inputs and outputs in Figure 4.8. Only connection slots having the same data type (thus having the same shade of gray) can be interconnected and vice versa. Processing pipelines are simply built by connecting NPGM boxes. Loops are allowed and can be exploited for feedback loops, e. g. for operators that change their parametrization according to results of subsequent operators.

The purpose of the box-related deployment constraints is twofold. First, they identify possible box implementations, being either a source operator, a sink operator or an operator. Second, they define on which NexusDS nodes the operators are executed. Besides deployment constraints also runtime constraints can be defined for each box. Thereby the possible runtime constraints are either specified by the concrete box implementation if specified on a physical level. Alternatively, if specified on a logical level, the possible runtime constraints are specified by the common subset of all runtime constraints a certain operator type defines to which the operator under concern belongs to. Defining a box on a physical level means, the actual implementation is specified. In contrast to this, defining a box on a logical level does not uniquely identify a certain box but rather defines a certain type the corresponding implementation of a box belongs to. This fact is depicted in Figure 4.8. Hereby, deployment constraints on a physical level are shown in **bold** letters (e. g. StreamNode='**SNx007**'), whereas deployment constraints on a logical level are shown in *italic* letters (e. g. Operator_Type='Render'). The logically defined deployment constraints must be mapped to deployment constraints that uniquely identify the box and the corresponding NexusDS nodes. In the following for boxes whose deployment constraints are specified on a logical level are referred to as logical NPGM boxes. In contrast to this, boxes whose deployment constraints are also specified on a physical level are referred to as physical NPGM boxes. However, it is important to note that physical NPGM boxes do not necessarily provide full deployment information as a (physical) NEGM box does. Physical NPGM boxes partially define deployment constraints on a physical level. Based on Figure 4.8, the four different box types are described: *logical NPGM* box, *physical NPGM* box, *logical NEGM* box, and *physical NEGM* box.

Logical NPGM box *Box3* represents a logical NPGM box as the deployment constraints for this box are logical. This logical operator is of type 'Link&Map' and the respective author is 'Visual Pipe'. The NexusDS nodes going to execute the operator must provide a 'Secure' execution environment which might be a domain-specific constraint as described in Section 4.1.

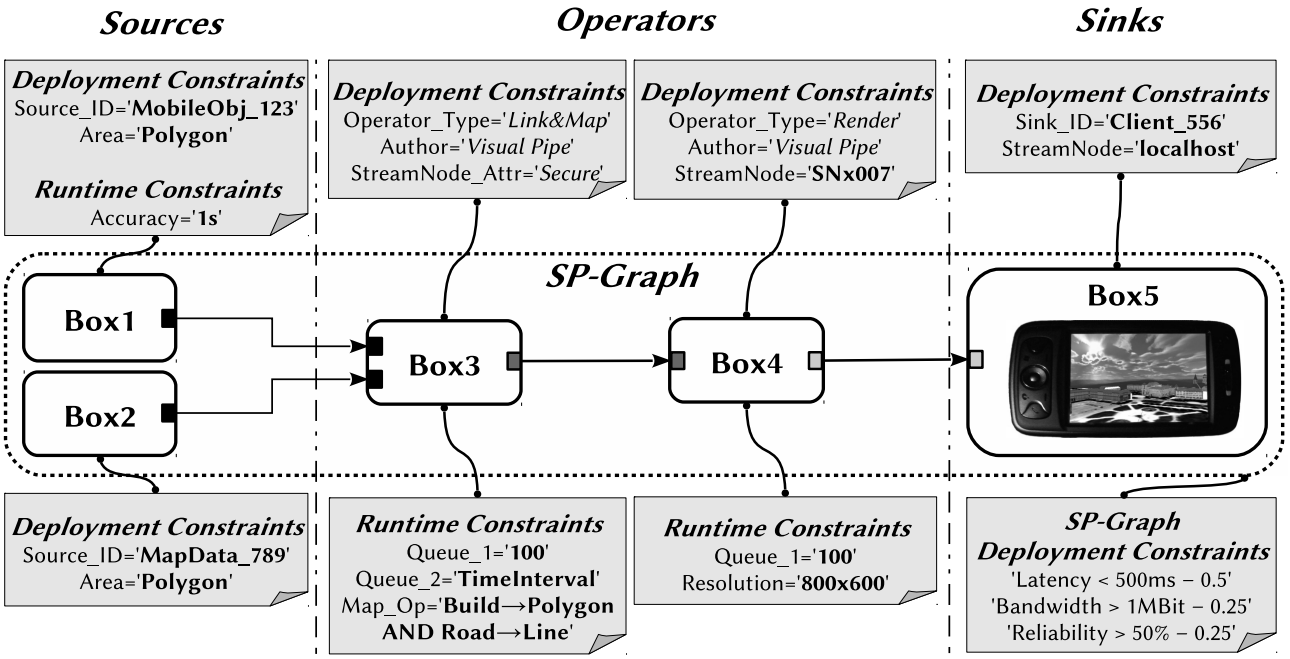


Figure 4.8: Constraint-aware NPGM SP-graph

Physical NPGM box *Box4* depicts a physical NPGM box. In contrast to a logical NPGM box either the physical operator or the physical NexusDS node are explicitly defined. As for the previously described Box 3, this box logically defines the operator being of the type 'Render' from the author 'Visual Pipe'. Here the NexusDS node going to execute the physical box is already specified as being '**SNx007**'. Nevertheless, since the operator is logically defined, it is not deployable and executable.

Logical NEGM box A logical NEGM box is represented by a box where the physical operator as well as the NexusDS nodes going to execute the physical operator are provided on a physical level. However, a logical NEGM box might provide many physical operators and physical NexusDS nodes. A logical NEGM box is not explicitly shown in Figure 4.8.

Physical NEGM box The difference between a logical and a physical NEGM box is that a physical NEGM box has exactly one physical operator and exactly one physical NexusDS node. In Figure 4.8, *Box5* represents a physical NEGM box.

Beside the deployment constraints shown in Figure 4.8, which specify the deployment constraints either logically or physically, the operator relevant requirements (described by the requirement meta data as presented in Section 4.2.1.2) must be added. They also constitute deployment constraints as they influence the NexusDS node selection and thus influence the deployment process. These operator-related constraints originating from the operator meta data are not shown in Figure 4.8. Nevertheless, they are necessary to find matching NexusDS nodes capable of executing one particular operator.

Beside deployment constraints also runtime constraints exist which influence the runtime behavior of boxes. However, the definition of these constraints is optional since presets are defined for each box, guaranteeing the correct execution. Preset parametrization can be over-

ridden as many users may have different preferences such as the rendering resolution for the rendering operator. The available runtime constraints depend on the actual operator, too. E. g., for join operators it is likely to set a join predicate whereas for a rendering operator it is usual to set a resolution.

There also exist **SP graph Deployment Constraints** as depicted in Figure 4.8. Application developers must provide these deployment constraints. Beside the other deployment constraints already mentioned, these constraints are exploited by the deployment framework presented in Chapter 6 to find suitable placement decisions for the boxes of a SP graph, i. e. source operators, sink operators, and operators. Thereby the box-related deployment constraints are exploited to reduce the possible search space to find deployment mappings in. Then, the SP graph-related deployment constraints are used to fine tune the deployment of the boxes according to quality aspects.

The SP graph-related deployment constraints are represented by a list of QoS requirements. These QoS requirements are fourfold. First, a QoS criteria is defined, e. g. *'Latency'* or *'Bandwidth'* from Figure 4.8. These criteria must be supported by the DSPS, i. e. QoS-related statistics must be collected as described in Chapter 6. Second, a case distinction must be provided which states if the QoS criteria is to be maximized ('>') or minimized ('<'). Third, a bottleneck condition must be defined. This bottleneck condition defines an absolute lower or upper bound value for the respective QoS criteria. If the QoS criteria is to be minimized, the bottleneck condition represents an upper bound. Otherwise it constitutes a lower bound. Finally, a relative importance factor between 0 and 1 must be provided for each QoS criteria, e. g. *'0.5'* for latency. They put each QoS criteria in relation to each other by means of importance. The relative importance factors must sum up to 1. The SP graph deployment constraints are valid for the entire SP graph and must be valid for each box. However, it is important to note that there are two different QoS classes we have to distinguish: *absolute* and *additive*. Absolute QoS criterion must be valid for each box. In contrast to this, additive QoS criterion must be valid in sum along the SP graph's critical path. This means that it is not enough for the additive QoS criterion to be valid for each box since the entire path is of importance.

4.6 Matching Deployment Constraints and Runtime Constraints

Once all constraints have been collected, i. e. requirement and capability constraints, they must be matched. At this point, we mainly differentiate between deployment and runtime constraints. Thereby, deployment-related requirement constraints are matched against deployment-related capability constraints. The same is performed for runtime constraints. Both matching types are briefly presented as follows.

To successfully deploy an operator, NexusDS needs to match the operator requirements with the available NexusDS node capabilities, i. e. its capability constraints. Therefore, the

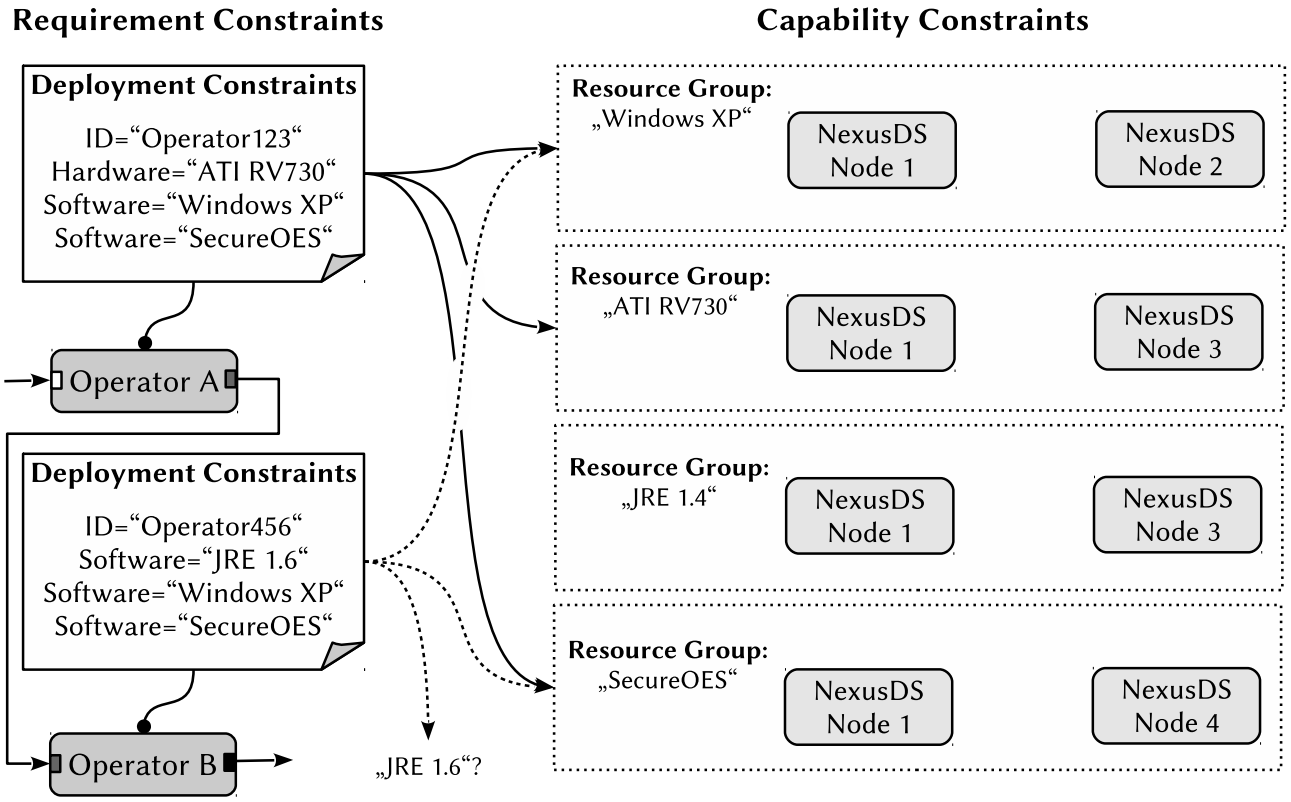


Figure 4.9: Matching the deployment-related requirement constraints with the available deployment-related capability constraints.

NexusDS nodes are grouped into resource groups to describe the capabilities of the respective NexusDS nodes. This enables operator virtualization. By virtualizing the operators, developers of stream-based applications do not have to care about the features of certain NexusDS nodes. This is handled by matching the deployment-related requirements of the operators with the deployment-related capabilities of the NexusDS nodes as depicted in Figure 4.9.

In the depicted example, a matching for *Operator A* can be found, since *NexusDS Node 1* is present in all relevant resource groups, i. e. *Windows XP*, *ATI-RV730*, and *SecureOES*[2]. This means that this operator can be successfully executed on this specific execution node, i. e. *NexusDS Node 1*. In contrast to this, for *Operator 2* no matching can be found. For the requirements *Windows XP* and *SecureOES* there are NexusDS nodes satisfying these. But there is no NexusDS node that satisfies the requirement *JRE 1.6*.

Also the runtime-related requirement constraints must be matched as depicted in Figure 4.10. Therefore, the *runtime constraints* of the SP graph are matched against parameters of the respective operator (displayed as solid arrows). These parameters are defined by the operators' descriptors. The values to which the parameters should be set to are also validated. E. g. the parameter *Parameter_1* of the operator with *ID="Operator456"* is set to **720x640**. If the possible values for this parameter have been restricted by the operator developer—which corresponds

[2] *SecureOES* represents the term of a secure execution environment for operators.

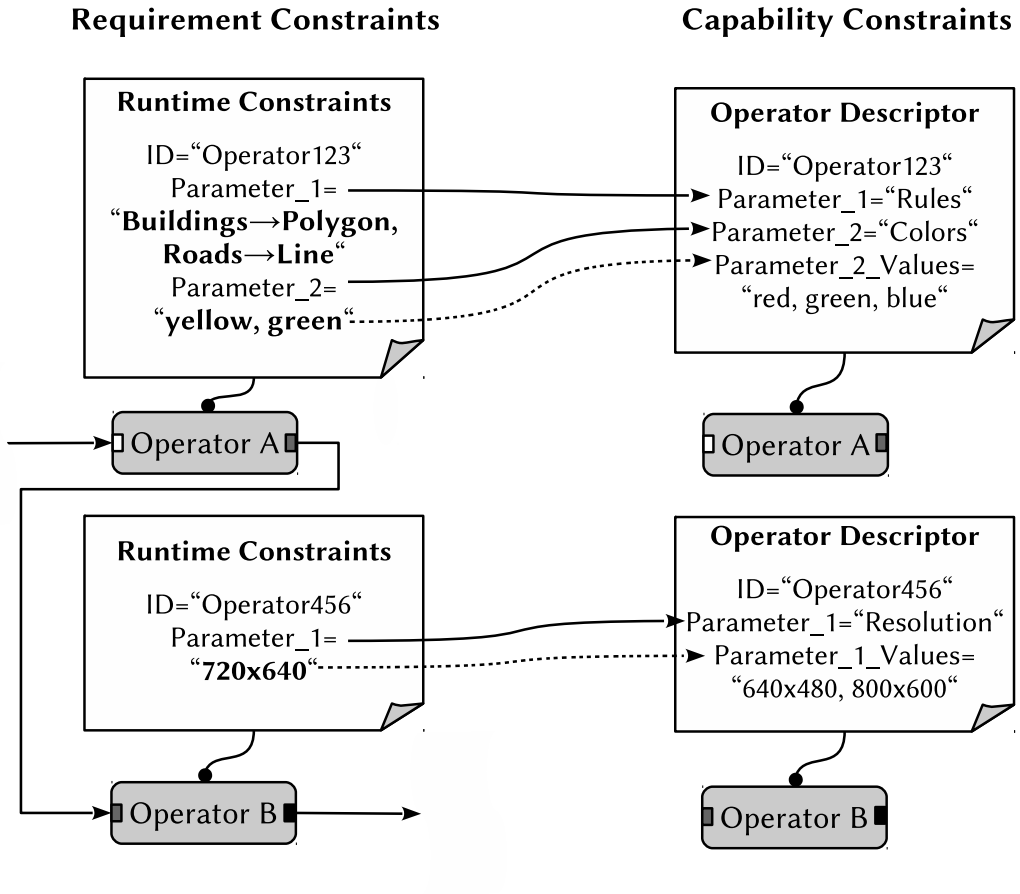


Figure 4.10: Matching the runtime-related requirement constraints with the available runtime-related capability constraints.

to the bounded modifiable constraints as described in Section 4.1.1—the value on the left side must appear in the list of values on the right side of Figure 4.10 (depicted as dashed arrows). Otherwise the operator cannot be executed with the given runtime constraints. Alternatively, if no values are predefined for a parameter, its value may be arbitrarily set. This corresponds to the class of freely modifiable constraints from Section 4.1.1. The limit of possible values is defined by the underlying components such as software drivers or the hardware itself.

The validation of the runtime constraints further restricts the possible NexusDS nodes going to execute the operators. Thus, this process directly influences the SP graph related deployment. E. g. if an operator rendering a scene is set to a resolution of **800x600** pixels, the corresponding NexusDS node should be able to provide these features by means of available memory.

4.7 Source Data Management

With the growth of online accessible data and information systems, the need for integration architectures is increasing. As can be seen in the Web 2.0 trend, more and more information is

provided by data sources, such as web sites, Wikis, or web services. It is a very cumbersome, and, on the whole, an almost impossible task to integrate this information for application use uniformly. However, when focusing on a certain application domain, we can exploit common characteristics to provide integrated views: e. g. WWW search engines integrate their search results based on rankings that represent the relevance to the user's query. We target on context-aware applications which support users with the right information at the right time and right place, i. e. providing the best information according to the user's current situation [51]. They often rely on large-scale information systems, where the data is scattered across a multitude of data sources ranging from web sites over digital libraries and geographic information systems to sensors and other stream-based sources.

The efficient retrieval and integration of the relevant information and the support of stream processing is a big challenge. This section presents a solution to the problem of incremental data retrieval in a federated environment, where in contrast to conventional distributed DB systems the actual partitioning of the information is not known a-priori and might change dynamically. There is an open world assumption, since data providers can dynamically connect and disconnect from the Nexus[3] system. Also, there can be multiple representations of real-world entities provided by several data providers. Nexus' open federation differs from conventional federated DB systems: It is based on simple object retrieval and does not provide the full-fledged SQL function set. Furthermore—from a conceptual point of view—it does not necessarily have to materialize the whole result set within the federation layer when integrating the results from the data providers.

The latter characteristic allows us to develop a scalable algorithm for object retrieval that works on partial results from data providers. Therefore, a federated cursor concept has been developed to allow piece-wise resultset retrieval and furthermore allow the integration of static data into DSPSs. Cursors are a long-known database concept that allows the application to piece-wise retrieve tuples of a result set [49]. This is especially beneficial if the applications run on resource-limited devices which typically retrieve information over expensive wireless communication channels. Such devices are often used in the areas of location-based services and pervasive computing. Furthermore, the cursor concept can be exploited to integrate static data in the context of stream processing, as NexusDS does.

Todays DSPSs are capable of handling both types of data, static data and dynamic (streamed) data. NexusDS provides access to streamed as well as static data. As presented in Section 2.5.3, NexusDS along with Nexus forms a context data management platform. NexusDS provides access to data stream processing capabilities whereas Nexus provides capabilities to query and process static data. As depicted in Figure 4.11, NexusDS exploits the functionality of Nexus to access static data by utilizing its federated cursor [41]. NexusDS implements a source operator that issues a cursor query to the Nexus system and retrieves the resulting data via cursor iteration. The Nexus system, more precisely the Nexus federation, distributes the query received to the relevant data providers which create so-called *cache histograms*. These query-related

[3]Recall: Each time we talk about the Nexus system we refer to the platform handling static data information as presented in Section 2.5.2. In contrast to this, NexusDS handles data streams.

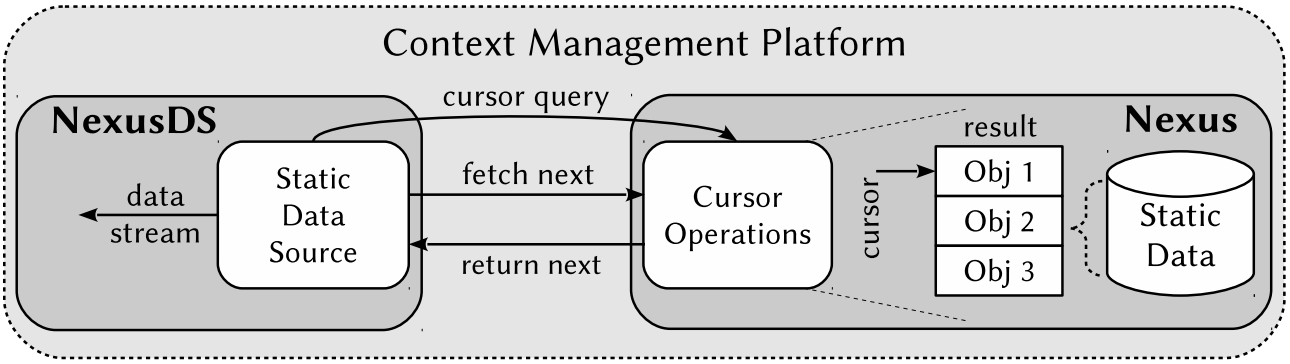


Figure 4.11: NexusDS accessing static data through Nexus.

cache histograms are sent back to the Nexus federation which collects and integrates them to a single one. A cache histogram is an ordered set of object references to the corresponding data providers. For one object in the cache histogram, multiple references to different data providers having a representation of the same object might exist. To get a complete object representation, it is important to retrieve all relevant object instances from any data provider. The integrated object is delivered to NexusDS and pushed into the processing pipeline as a data stream. The data stream ends if either data retrieval is complete and thus there are no more objects to fetch or if the processing of the related SP graph terminates. The details on the federated cursor concept are provided in Section 4.7.2.

4.7.1 Related Work

There has been some work addressing the problem of efficiently processing and incrementally retrieving partial results. Haas et al. [64] propose to speed up data intensive applications needing fine-grained object access by loading the cache of the system with relevant objects. The decision of what objects are relevant is made by the frequency applications access objects. However, this technique does not consider multiple representations of the same object containing incomplete or partial information distributed over several data sources. In this case one has to find and fetch all representations of an object in order to get a complete and consistent object representation.

Garlic [73] is a platform for federated data management of relational data sources based on IBM DB2. For the incremental retrieval of the result set two possibilities are mentioned. One is to materialize the entire result of each data source. The other is the retrieval of data using the cursor mechanism. Each time *Fetch*[4] is invoked, one data element at a time is retrieved from the data source. Thereby, no possibility of sophisticated retrieval of the result sets is mentioned. The possibility of incomplete partial results is also not taken into consideration in this approach.

In Disco [136], the problem of dealing with unavailable data sources is addressed. The selected approach uses a *partial evaluation* semantic to return partial answers to queries. Here

[4]This is a cursor operation to iteratively fetch data elements from a result set.

the parts of the query that could not be answered are mapped back to Object Query Language (OQL) [5] and integrated with the parts of the query being answered by data providers. It is neither described how exactly the results are retrieved from the data providers nor how the partial results of the portions of the query are integrated into a single answer.

Information Manifold [86, 87] deals with the efficient query processing in a distributed environment that involves a large number of data sources. They use descriptions of the data sources for a given query to identify relevant sources, query these sources and finally collect the complete result from these partial results. The query processing engine tries to recognize sources providing redundant information and prunes them. No integration of the partial results or further computations are made. This has to be done by the inquiring application. Also, no further reflection on alternative retrieval mechanisms are made.

There also exist several mediator-based systems like TSIMMIS [55] or MedMaker [109]. However, we focus on context-aware applications using context-data, and furthermore provide domain-specific operators and optimizations.

4.7.2 Federated Cursor Concept

The main characteristics of the federated cursor concept are

- to request only context servers that actively contribute to the given query,
- to process only the resulting data which is necessary,
- to support temporarily disconnected applications (in this sense the source operator is an application), and
- to integrate static data into stream processing systems.

The original idea of a cursor is to bridge the so-called *impedance mismatch* between the set-wise processing of data in DBMSs and the tuple-wise processing of data in programming languages. Thus, the cursor allows programming languages to cope with tuple-wise processing by providing a pointer to the actual tuple to be worked up.

This idea has been adapted to the domain of our federated and context-aware platform. Here, the cursor concept is used for retrieving partial results of queries in order to prevent memory overflow and to save communication bandwidth, thus bridging the *resource mismatch* between often resource-limited mobile devices and the resource-rich server infrastructure. Furthermore, our federated cursor concept is exploited by source operators in NexusDS providing static context data. E. g. by using a cursor, an application does not have to wait until the entire result is transferred before processing it. Depending on the type of connection there may be unwanted disconnections: E. g. the larger the result, the higher the risk that the full result never reaches the querying application.

In the subsequent sections the concept of a federated and status-conscious cursor is introduced. The cursor is used to efficiently retrieve objects over distributed data sources.

[5] **http://www.odbms.org/ODMG/**

4.7.2.1 Cursor-based Query Processing Sequence

As displayed in Figure 4.12, a Nexus application posts a query and receives an answer containing the query result. A Nexus application is an application which runs on the Nexus system (see Section 2.5.2). From a NexusDS point of view, a special form of Nexus application is the source operator that allows to integrate Nexus data into NexusDS' stream processing facilities iteratively.

The cursor-based processing model consists of three phases. This is analogous to the cursor processing as described in [49]. The application has to post a query associated with a cursor on the query's result. After that, the application can start to gradually process the result. In the end, the result is deleted if its lifetime has expired or the application signals that it is no longer needed.

Phase 1: Initialization Phase During the initialization phase, preparations for the next phase are performed. The necessary steps are as follows (see Figure 4.12):

(1) An application sends a cursor query to an arbitrary Nexus node asking the federation to create a cursor on the query's result.

(2) The Nexus node determines the relevant Context Servers (CSs) by an Area Service Register (ASR) lookup based on the spatial restriction and the object type in the query.

(3) The Nexus node forwards the query to the CSs which process the query and send their results back. Additionally, the federation sends back an ID, a so-called Nexus Session Locator (NSL),—details see later—of that cursor to the application.

Phase 2: Delivery Phase If the initialization phase has been successfully completed, the application is able to send cursor operations on its cursors to the federation to give access to the result data piece by piece. This phase is called delivery phase. The necessary steps are as follows:

(4) The application posts a *next* operation stating the next elements pertaining to a certain result which is identified by the NSL.

(5) The federation looks for the results belonging to the NSL and prepares the objects that go to the result set. Objects have to be retrieved from the CS (if they are not already in the cache).

(6) Multiple represented objects (MReps) have to be detected and merged. This operation may reduce the effective number of objects.

(7) The result set is sent back to the application.

The application repeats the delivery phase until end of resultset is reached or it does not need any further elements and decides to finish the retrieval process. In both cases, the federation enters the termination phase.

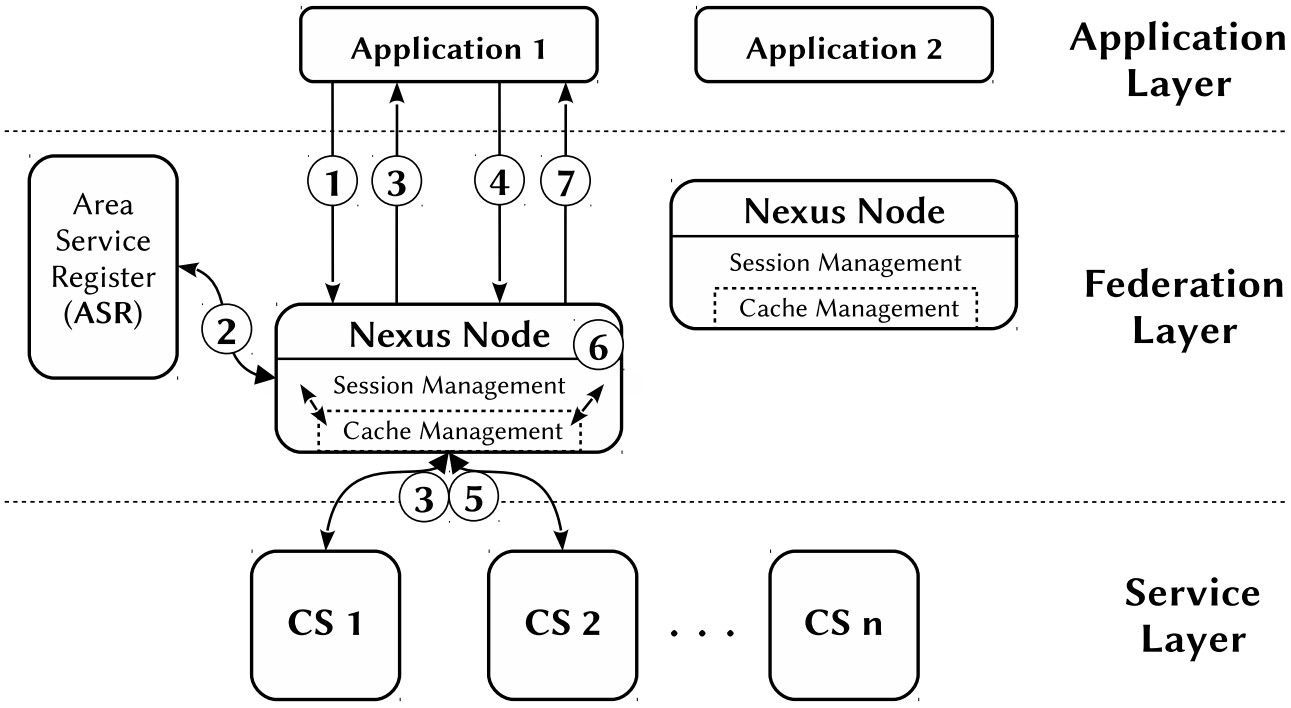


Figure 4.12: Cursor-enhanced Architecture of Nexus.

Phase 3: Termination Phase This final phase is entered if the lifetime of the result has expired or if the application signals to the federation that it does not need more elements. The resources connected to the ID are released.

4.7.2.2 Session Management

For identifying sessions within the Nexus platform a so-called NSL is used. An NSL consists of two parts: a basic service part which encodes the Nexus node the session was created on and thus holds the session information, and a session ID. The hosting node is encoded within the NSL to support distributed session management. Since we want to support mobile devices, an application can switch its Nexus node. Using the NSL, a Nexus node that receives a cursor query for a cursor that does not run on that node can easily forward that query to the correct node. If a mobile device switches the Nexus node during operation, the new node has to retrieve the specific application information from the relevant Nexus node encoded in the NSL in order to be able to process the request adequately. For this, there are two possibilities: One is to transfer all relevant information to the new node and to replace the host address in the NSL. The alternative approach always forwards the query to the original node.

4.7.3 Federated Processing Strategies

After the introduction of the federated cursor concept, we investigate the federated processing strategy for incremental result retrieval. In order to optimize query processing, the Context

Server (CS) should support a cursor concept. However, this is an optional feature of a CS. Without that functionality, the entire result of each CS has to be transferred to the federation. In Figure 4.12, the CSs are also extended by a session management in order to be able to hold application specific information. In that way, the results can be kept locally at each CS and only objects that are needed will be transferred to the federation.

To provide optimal response times for applications, a federated DB system should pre-cache partial results [64]. There are several ways to do this in the federation. The naïve approach is to query all relevant CSs and cache all results locally. This approach has the advantage that there is no more communication overhead between the federation and the CSs, and long latencies for query answering are avoided. But it suffers from high memory consumption within the federation layer and a long initialization phase since the results of all CSs must be fetched first.

A more sophisticated approach is to reduce the memory consumption at the federation layer. Therefore, spatially divided queries can be sent to the CS. Here the initialization phase consists of the non-trivial problem of partitioning the query. Objects may be queried that currently are not needed and in a worst case scenario never will be. Furthermore, care must be taken that all MReps are present for the merge operation at processing time.

Both solutions sketched above are not recommendable. The former suffers from high memory consumption in the federation layer, while the latter suffers from communication overhead between the federation and the CSs and could also miss information for some objects. So, there is a trade off between memory consumption and the system load.

For the remaining discussion a spatial partitioning scheme is assumed, which Nexus naturally supports. This influences the initialization phase described above as the query partitioning changes. However, the federated cursor concept is also applicable for alternative partitioning schemes. As the query partitioning is transparently done, e. g. by a federation layer, it has no consequences for the further federated cursor details as presented in the following.

4.7.3.1 Cache Histograms

A major feature of the Nexus federation is the ability to manage and merge MReps. To ensure that this operation is performed correctly in the cursor mode, we have to pre-cache partial results in a way that all candidates for a MRep merge operation[6] are present whenever this operation is carried out. The naïve way would be to pre-cache the whole resultset at federation level. However, this introduces the often unnecessary memory usage at federation level and possibly even a communication overhead between federation and context servers, e. g. if an application does only retrieve partial results.

Cache histograms allows to solve this problem in an efficient way. A single cache histogram represents the query-dependent frequency distribution of the resulting objects based on a sorting criterion, i. e. the distance from a geographical point. Cache histograms are provided by

[6]This operation merges all representation os the same real world object and is described in [121].

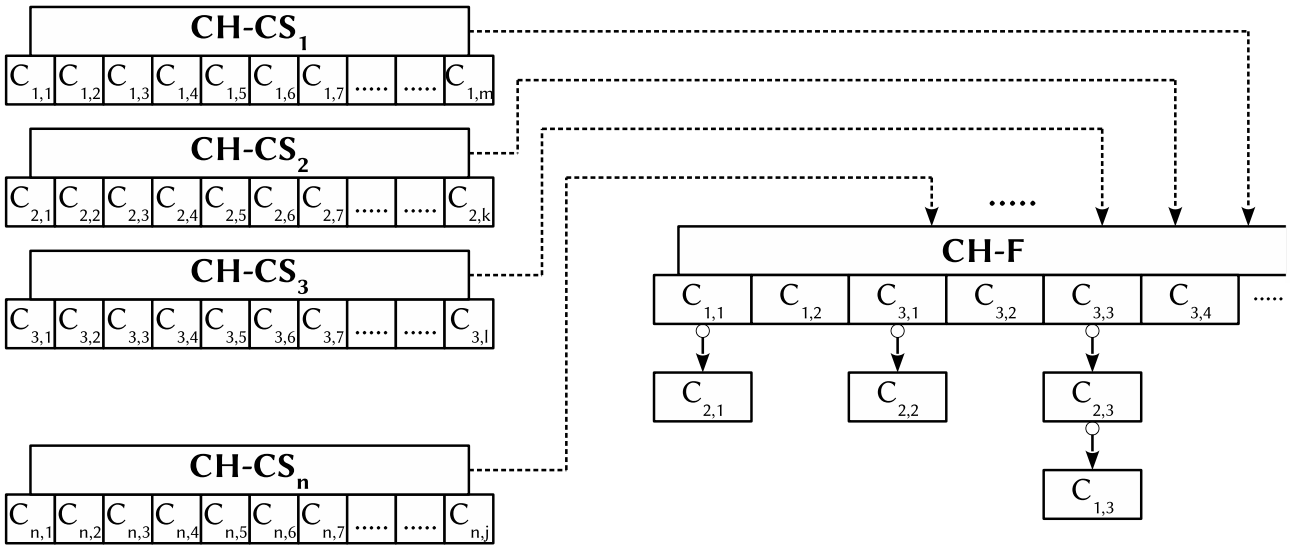


Figure 4.13: Federated cache strategy using cache histograms.

each CS. A cache histogram consists of a set of cache histogram entries. Each cache histogram entry consists of a bucket value which indicates how the partial result of a context server was sorted and the amount of occurrences of that bucket within that partial result. A bucket here refers to a discrete point in the sorting domain and not to an interval as usual.

As shown in Figure 4.13, each CS delivers a cache histogram (**CH-CS$_1$** to **CH-CS$_n$**) which is spatially sorted. **C$_{a,b}$** corresponds to a cache histogram entry and gives the value of the cache entry and its frequency of occurrence. In our example, the value refers to the distance of an object to the reference point. E. g. **C$_{1,1}$** is the first entry of the cache histogram for CS **1**, where **C$_{1,2}$** would be the second and so forth. The entry **C$_{1,1}$** has the value **<17, 5>**, which addresses the **5** nearest objects from **CS$_1$** with a distance of **17** to a reference point. Usually this reference point constitutes the user's current position.

If cache histograms supplied by the corresponding CSs are not already sorted by the bucket value, the federation has to do it by itself. That might occur if the cache histogram is created before sorting the partial result or the CS does not support sorting at all. However, the federation merges the cache histograms delivered by each CS into a federated cache histogram in order to get an overall overview **CH-F** of all data sources involved in the incremental retrieval process. The most important information at this point is which CSs have to be queried, the order in which the CSs should be queried, and the quantity of objects (which is encoded in the cache histogram entries) to query the CSs for.

Since there may be multiple representations for the same real world object, there can be objects with the same sorting value in different cache histograms. In this case, these entries are stored as a linked list, as shown in Figure 4.13. Elements in the linked list potentially represent the same object. All the objects belonging to the same list must be transferred to the federation in order to ensure a lossless merge of the intermediate results. The federation's merge algorithm decides whether two or more entries in the linked list represent the same

object. In the example depicted in Figure 4.13, the objects $\mathbf{C_{1,1}}$ and $\mathbf{C_{2,1}}$ got the same bucket value and are thus stored as a linked list. Taking for example $\mathbf{C_{1,1}}$ with a value of **<17, 5>** and $\mathbf{C_{2,1}}$ with a value of **<17, 3>**, the federation would first ask $\mathbf{CS_1}$ for the next **5** objects and then $\mathbf{CS_2}$ for the next **3** objects. Each of the objects have a distance of **17** to the reference point.

Listing 4.4 shows the cache histogram algorithm in pseudocode. It is used by the federation to build up a federated cache histogram **CH-F**.

4.7.3.2 The Retrieval Process Using Cache Histograms

Internally, the cursor is split in a horizontal **H** and vertical **V** component. The **H** component traverses the cache histogram from left to right in sorting order. The **V** component goes from top to bottom, i. e. by linked objects for a certain sorting value. Figure 4.14 shows a *next* operation on the federation cache histogram. The initial state of the algorithm is displayed in the upper left. The **H** component corresponds to the current cursor position. The **V** component indicates the position within the linked list of elements with the same bucket value.

When performing a *next* operation, first all elements in the linked list have to be processed ①, to ensure that all representations of the same object are retrieved. The next step is to move the **H**-position one step to the right ② to prepare the following next operations. These steps are repeated until the algorithm has retrieved all objects needed to answer the current next operation. The retrieval process ends when the application decides to stop retrieval at any time or if there are no more objects to retrieve.

```
1   // application sends query to system
2   receive application query
3
4   // determine the relevant context servers and send answer
5   ask ASR for relevant context servers
6   send NSL to application
7
8   // send query to all necessary sources
9   for each context server do
10      // get cache histograms from each context server
11      forward query and receive the cache histogram
12      // eventually sort them
13      if cache histogram not sorted
14          sort cache histogram
15
16  // merge the cache histograms
17  merge cache histograms to federated cache histogram
```

Listing 4.4: The cache histogram algorithm.

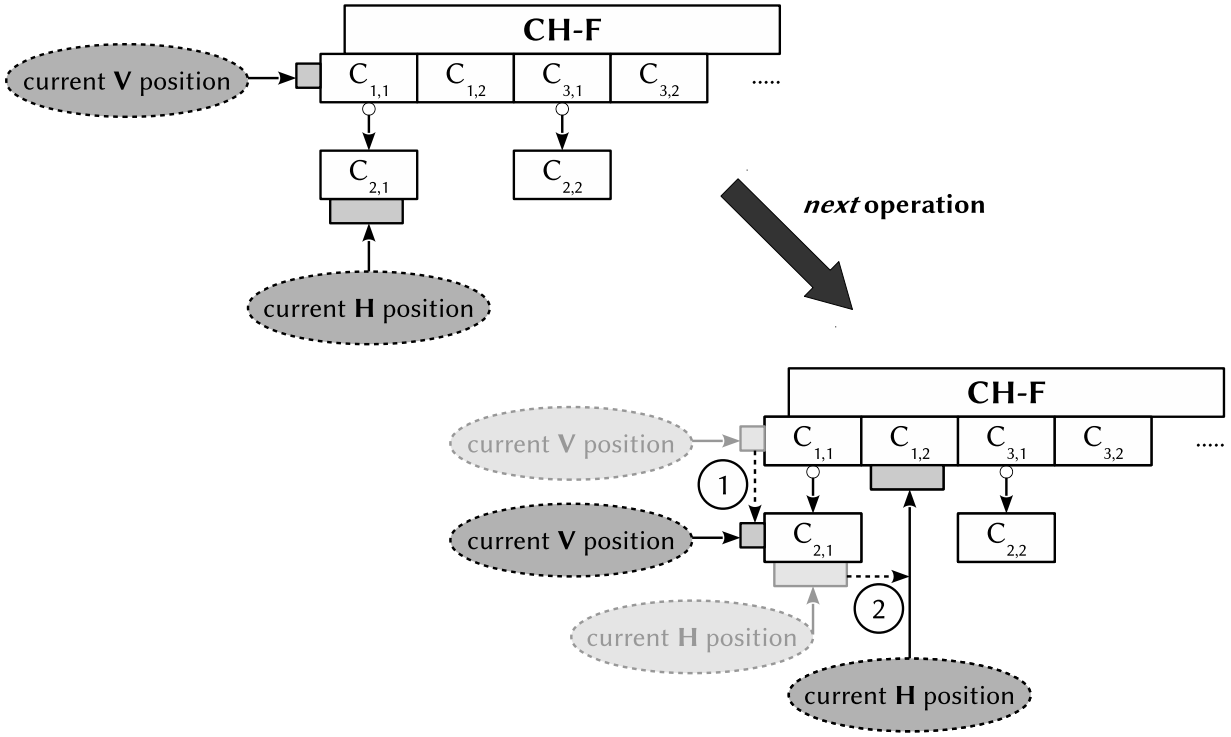


Figure 4.14: One cache histogram retrieval step.

The algorithm works in an efficient way in terms of memory consumption and network load because only relevant CSs are queried for objects. Furthermore, no MReps are missed out. Listing 4.5 shows a simplified version of the retrieval process using cache histograms in pseudocode notation.

```
1   // application requests next N objects
2   K := number of objects in output buffer
3   PL := []
4   do N − K times
5       // output buffer does not contain enough objects
6       if V−component points to cache histogram entry
7           P := context server in current cache histogram entry
8           M := bucket size in current cache histogram entry
9           if P in PL
10              increment number of objects to fetch from P by M
11          else
12              append P to PL
13              set number of objects to fetch from P to M
14          move V−component one step down
15      else
16          move H−component one step right
17  OL := []
18  for each P in PL do
```

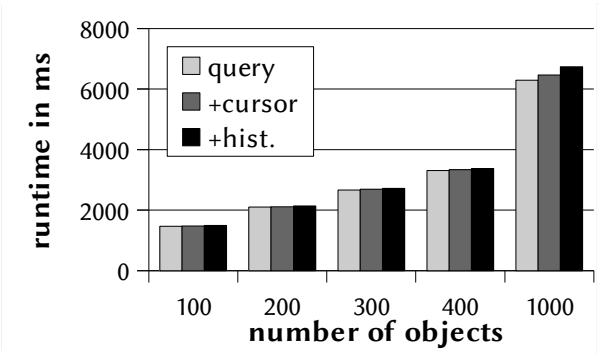
```
19          retrieve the given number of objects from P
20          append objects to OL
21      merge objects in OL
22      append OL to output buffer
23      remove first N objects from output buffer
24      send removed objects to application
```

Listing 4.5: Retrieval process algorithm using cache histogram.

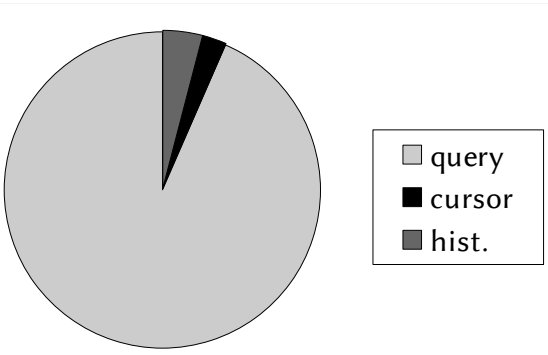
4.7.4 Experience and Evaluation

Considering scenarios in which mobile devices must retrieve result sets piece by piece due to memory limitations of the device, the cursor concept is an advantage. Furthermore, the cursor functionality can be exploited by NexusDS source operators as the federated cursor concept preserves resources and thus allows to serve many different applications. For these reasons the Nexus system has been enhanced by a federated cursor concept as presented in this section. A series of experiments have been conducted which show that the additional overhead caused by the cursor management and histogram calculations is comparatively small.

The context server used for the experiments is implemented in Java and was running on a SUN Blade 2000 with two 1.2 GHz UltraSPARC III CPUs and 6 gigabyte (GB) of RAM. IBM DB2 8.1.3 was used as the backend data storage system. The DB contained 3380 AWM objects in total. Figure 4.15(a) shows the runtimes of a nearest-neighbor-query with sorting by distance from a reference point. We varied the number of objects to retrieve between 100 and 1000. The measurements for *query* refer to query processing alone, *+cursor* additionally creates a cursor and *+hist.* furthermore computes a histogram. Figure 4.15(b) shows the fractions of the runtime required for processing the query, creating a cursor and computing a histogram for the 1000 objects query. The extra overhead is below 7%. This fraction is even lower for smaller



(a) Runtime measurements for different result set sizes.



(b) Runtime fractions for cursor management and histograms for a 1000 objects query.

Figure 4.15: Runtime measurements.

result sets, approximately between 0.5% and 2%. The overhead grows linearly with increasing resultset size.

The positive effect of the cursor and the histogram gets clear if we consider the following two facts: First, the cursor allows to interrupt or even stop result retrieval at any time since we may not want to retrieve the entire result. Second, the histogram enables the efficient and complete retrieval of AWM objects necessary for the next processing step. This shows that the overhead introduced by the cache histograms and the cursor functionality is relatively small and therefore negligible compared to the query processing itself.

4.8 Summary

This chapter introduces a constraints classification in the domain of DSPSs. Two basic constraint types are differentiated: Deployment constraints and runtime constraints. The constraints occur on different levels. I. e., an application user may formulate user-relevant constraints. The application itself can also formulate application-relevant constraints and so forth. These constraints are propagated by annotating the SP graph by the respective constraints. By these constraints the deployment and runtime behavior of a SP graph can be influenced. The operator and service model have also been introduced and their particulars highlighted. With the flexible models presented it is possible to implement and provide arbitrary functionality. At the same time, a tight integration with other system functionalities can be achieved. Related to the operator model and the service models, the processing model of NexusDS has been introduced. Finally, the characteristic handling of static data originating from Nexus has been introduced. Hereby a sophisticated federated cursor is introduced which allows source operators in NexusDS to retrieve efficiently objects from the Nexus system and make them available for further stream processing operations. The next chapter deals with details on the security concept to control access to data as well as influence the processing of data according to security policies. These policies are integrated into the SP graph, making it compliant to security policies.

Part III

Security Framework and Stream Processing Graph Deployment

5 Security Framework

After the presentation of the NexusDS architecture in Chapter 3 and the processing issues in Chapter 4, in this chapter security aspects in DSPSs are introduced and discussed. NexusDS defines different security patterns, each one covering certain security aspects as defined by the requirements in Section 2.4.3. Next, in Section 5.1, a motivating introduction to the topic of security in DSPS is presented. Thereafter, Section 5.2 defines protection goals and provides a classification for access control and processing control mechanisms. Related work is discussed in Section 5.3 comparing the security-related aspects of the state-of-the-art DSPSs. In Section 5.4, the security control framework and its mode of operation is presented in detail. After that, the architecture of the security framework with all its main components is described in Section 5.5. This section also presents the security compliant extension to the operator framework introduced in Section 4.2. The security policies are handled by the DSPS at deployment time and may change during runtime as described in Section 5.6 and Section 5.7. Finally, this chapter is concluded by a short summary in Section 5.8.

5.1 Motivation

With the rapidly increasing popularity of mobile phones equipped with GPS sensors and mobile Internet connections, the usage of data stream processing is increasing in many application areas. A GPS sensor, for example, continuously produces a potentially unbounded stream of location information which makes the usage of data stream processing recommendable. Application areas for data stream processing can be found in social media applications—such as *Facebook*, *Twitter* and *Google+*—as well as in LBSs. A combination of both—social media and LBS—is the *foursquare* service, which can be defined as a *location-based social media network*. Therefore, location data is augmented with personal information. The benefits of LBSs is undisputed and already included in many of today's smartphone applications.

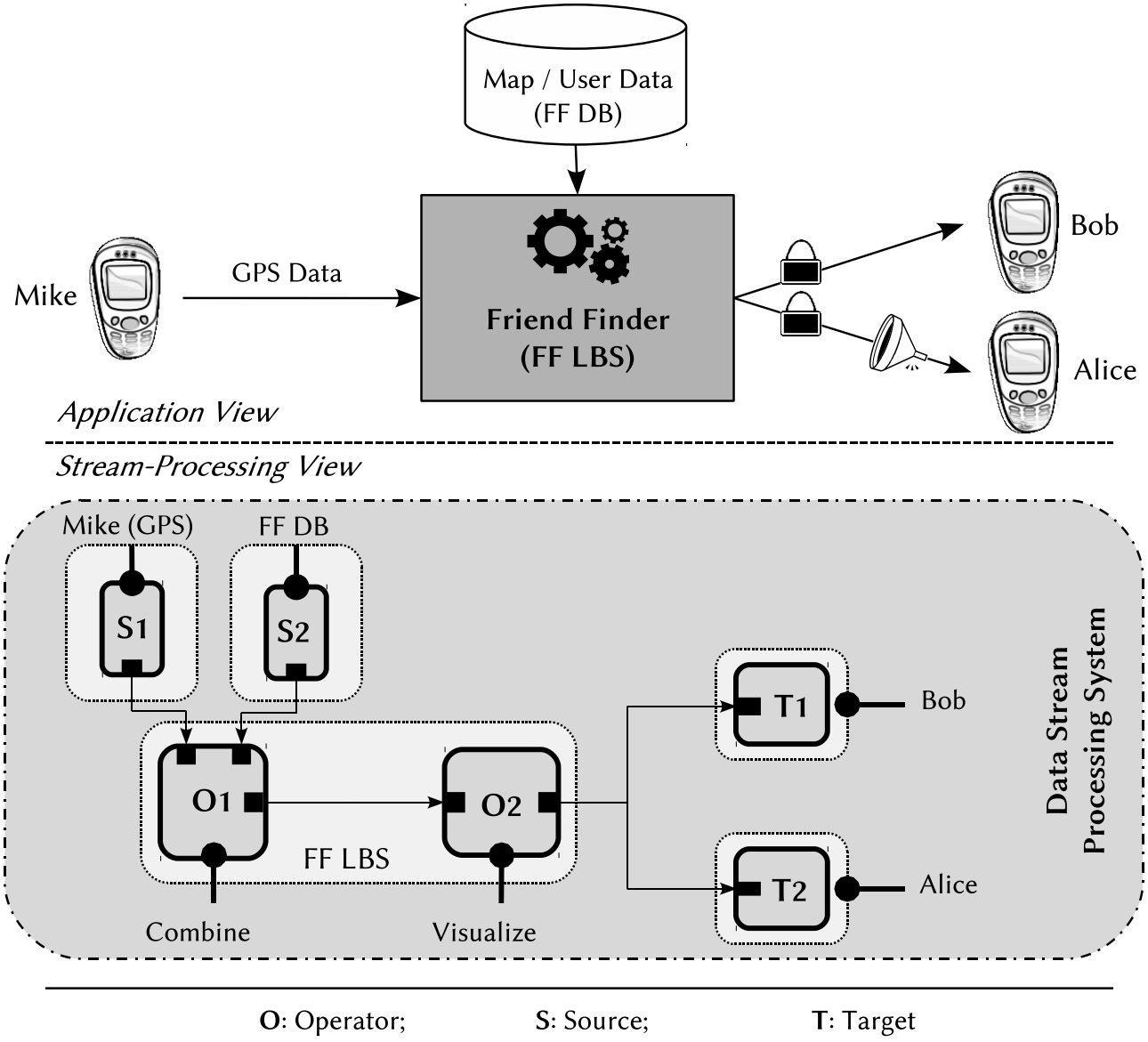


Figure 5.1: Application scenario illustrating the fictive location-based social media network data streaming service *Friend Finder*.

More and more of our social and private life is pervaded by LBS and social media applications, which on the one hand deliver a real benefit in everyday life providing location-based information which one might be interested in. On the other hand, this raises the question of how to protect information against unauthorized access. It is of great interest for the data owner to express fine-grained access conditions, defining which data can be accessed by certain entities and how this data might be processed by DSPSs. Social network services as well as LBSs are easy to use and information including personal details and the current position is available to a wide audience. This creates a variety of usage scenarios but at the same time exposes possibly sensitive information to the public.

The upper part of Figure 5.1 (Application View) depicts a fictive LBS, called *Friend Finder* (FF). This sample application reveals the current location of a user to all of his friends. Mike

broadcasts his GPS data to FF in our scenario. FF then combines his data with additional information acquired by third-party data providers, e. g., Google Maps. Similar services are offered by many of todays social networks such as *foursquare*[1]. However, through these services a user, i. e. Mike, can share all of his private information with a user or no information at all. In contrast, our approach goes a step further: Although both of Mike's friends Bob and Alice have access to parts of his data, Alice receives filtered information only. E. g. while Bob gets Mike's accurate location, Alice only gets the country where Mike is at the moment.

In the lower part of Figure 5.1 (Stream-Processing View) the participants of this scenario are mapped to the nodes of a DSPS. Mike's GPS and the third-party data providers act as data sources (**S1**, **S2**) while Bob's and Alice's mobile devices are the data sinks (**T1**, **T2**). The FF LBS processes this data (e. g. by combining different sources **O1** or by visualizing the data **O2**) and ensures that Mike's privacy settings are respected. In order to make these features work, the DSPS that performs data integration and data processing must provide access control to data [11]. Furthermore, it must provide fine-grained process control mechanisms.

A prerequisite to support a wide range of stream-based applications—including the application scenario sketched above and especially context-aware stream-based applications—is that the DSPS must be open to further application scenarios and must provide an integration mechanism for domain-specific extensions [42]. This means that the required domain-specific data and processing techniques—in terms of operators—should be integrated into an existing system to exploit functionalities already existing and extend the system only where necessary, reducing functional redundancy. As context-data is highly privacy-related, security control patterns are essential to control data access and data processing.

The openness of NexusDS in combination with the requirement of fine-grained data access and fine-grained data processing as described beforehand is a big challenge. Appropriate mechanisms must be developed to allow for both a controlled access to and a controlled processing of sensitive context data. For this purpose we extended NexusDS to meet these requirements. We have developed an access control framework for DSPSs which retains the openness and flexibility of the original DSPS and allows—depending on the desired level of security—to determine the settings for fine-grained data access and data processing.

5.2 Protection Goals

The definition of the access control framework for a DSPS is preceeded by the definition security requirements to data. Depending on this requirement catalog the *secure processing mechanisms* are designed. Section 5.2.1 first describes the necessary terminology. After that, in Section 5.2.2 the access control classification is shown to ensure—in the context of this thesis—*safe processing* by establishing a security architecture that allows to define access conditions and processing conditions of context data.

[1] **https://www.foursquare.com/**

5.2.1 Clarification of Terms

The access control framework distinguishes two types of participants: *Subjects* and *objects*. Subjects represent entities such as users of a system or a process running in a system. In contrast, objects represent entities such as files, database entries, or executable code within a system. Subjects access objects. However, in certain circumstances a subject may become an object. Imagine a subject that wants to change the access conditions of another subject, such as Mike from Figure 5.1. Mike wants to limit the position sharing to users labeled as members only. Throughout this chapter we refer to the respective entity type being either a subject or an object.

5.2.2 Classification of Protection Goals

Information can be protected under consideration of different protection goals, which influence the actual design and functioning of a system or process. Our classification is built on five protection goal classes which in turn consist of a variety of targets. The protection goal classes are: *Authentication*, *Access Control*, *Process Control*, *Granularity Control* and *Enforcement*.

- **Authentication**: It covers all goals for the reliable identification of the relevant subjects and objects which are participating in the system. These include the *authenticity* of subjects and objects which must have the necessary rights to join the system as well as the *action liability* which assigns each action to a specific subject. To make these actions traceable, a storage area for trace information must be provided.
- **Access Control**: It covers all goals concerning access control. Here the data *integrity* is of crucial importance. This ensures that objects cannot be changed uncontrollably and therefore guarantees that only subjects allowed to make changes will be able to access the requested objects. Furthermore, the *confidentiality* of information must be ensured in order to hide information from subjects who are not allowed to read this information.
- **Process Control**: It covers all goals influencing data processing. This includes the *acceptance* of computation environments which are going to process the data. Assuming a distributed environment, acceptance defines the computation nodes the data might be processed on. Besides this, also *data extent* is of importance, i. e. to define the amount of data that is available at one time instant for data processing. By limiting the data extent a limited view of the current data is provided.
- **Granularity Control**: It covers all goals concerning the obfuscation of the original data (object) in order to prevent conclusions to the subject the data originates from, with techniques such as anonymization and pseudonymization. Other techniques which belong to this protection goal class are methods that add some fuzziness to data in order to hide detailed information on current positions etc., or aggregate a certain amount of data elements before delivering it to subsequent operations.

- **Enforcement**: Defines how the security policies, i. e. access control, process control and granularity control policies are enforced. An important fact is the enforcement scalability. In a distributed environment the enforcement should not happen at a single site but should rather be done in a distributed manner to avoid bottlenecks or single point of failures.

These protection goals build the basis for the comparison of related work in the section that follows. The protection goals also define the main functionalities of our security framework which is the foundation of our system implementation as shown in Section 5.5.

5.3 Related Work

This section introduces some well-known security concepts in the context of DSPS and provides a comparison according to the protection goals raised in Section 5.2.2. A DSPS is characterized by an asynchronous and distributed execution of long running queries. This represents a major challenge since access policies might change at runtime which require the use of appropriate measures in order to ensure changed security policies being enforced. These changes should not influence ongoing operations as this would affect currently running queries negatively. On the other hand, new policies should be enforced as quickly as possible to avoid unwanted visibility of data while avoiding centralized structures, as they constitute a single point of failure. Table 5.1 provides a comparison of well-known concepts in this area w.r.t. the protection goals presented in Section 5.2.2. These single concepts are described in detail in the following. An overview of these systems can be seen in Chapter 3. However, in this chapter we focus on the security aspects and therefore wrap up the state-of-the-art of security mechanisms in the context of DSPSs.

In the year 2005, *Secure Borealis* [88, 89]—which extended Borealis [2]—was one of the first DSPS which had an integrated security concept to control data access. The security concept is based on a general DSPS architecture to which additional components that enable access control were integrated. The query processing in Secure Borealis is performed in a distributed fashion. Communication between the single computation nodes is encrypted to ensure data integrity and to prevent data from being read by third parties. In contrast to the query processing, the security concept of Secure Borealis is based on a centralized structure to enforce security policies. This circumstance is a potential bottleneck and represents a possible single point of failure since all data first has to pass through this component before it can be forwarded to subsequent operations or the target. This centralized component enforces access control and consists of two parts, an *Object Level Security* (OLS) and a *Data Level Security* (DLS) component. The OLS component is active before the runtime of queries and the DLS component is active during query runtime. The OLS component is linked to a role model that assigns to each subject (e. g. a user) a specific role that holds its associated access permission. Subjects are identified by a username and password combination. Based on this information the OLS component decides whether a subject is allowed to access objects (e. g. data). All ob-

System	Authentication		Access Control		Process Control		Granularity Control	Enforcement
	Authenticity	*Action Liability*	*Integrity*	*Confidentiality*	*Acceptance*	*Data Extent*		
Secure Borealis	Role	Subject	Encryption	Centralized Control	—	—	"All or Nothing"	Centralized Supervisor
ACStream	Subject	Subject	Predicate	Predicate	—	Time-based Windows	"All or Nothing"	Rewrite Queries
FENCE	Subject	Subject	Predicate	*SS+* Operator	—	—	"All or Nothing"	Rewrite Queries/ Security Punctuations
NexusDS	Role	Subject	Encryption/ Certificates	*SI* Filter	Computation Node Set	Parametrizable Windows	*LoD* Filters	Augment Queries/ Security Punctuations

Table 5.1: System comparison w.r.t. the protection goals authentication, access control, process control, the possibility of controlling the access granularity of context data, and how the protection targets are enforced by the respective system.

jects a subject is not allowed to access are hidden. The DLS component enforces the security policies at query runtime and consists of filters that are applied to the final result of the queries to remove objects (data elements) from the resulting data streams, to which the subject has no access. This in turn means that the access control enforcement is performed after the final data elements are determined, i. e. the entire query processing is done. This strategy might discard costly calculated data resulting in a waste of resources.

The access controls in *ACStream* [29] can be defined on a data stream level for data elements and their attributes. ACStream builds upon Aurora [3]. Access control restrictions are defined by expressions that describe an explicit assignment of access rights to certain subjects. To illustrate this, imagine a data stream holding positions where each data element has an ID-attribute and a location-attribute. An expression in ACStream can define read access for a subject *A*, if the ID-attribute has a certain value, or the position is within a defined quadrant. A special feature of the concept is the possibility to define temporal constraints. A temporal restriction allows access to data elements which are situated in a certain time interval. The start time and end time can be explicitly defined and the time interval is provided by defining the absolute time interval size and the interval step size. ACStream enforces access control by rewriting queries. The rewrite process selects *security operators* instead of regular operators to carry out the defined access control constraints. Four different types of security operators are available: *Secure View* processes an input stream by applying the access restrictions and returning a view of the data stream with only data elements meeting the access restrictions. *Secure Read* operators filter data elements and remove attributes, *Secure Join* operators filter the output data streams composed of multiple input data streams, and *Secure Aggregate* operators control aggregate functions.

FENCE [98, 99] uses *security punctuations* to enforce access control to data streams. A security punctuation[2] is a data element within a data stream that defines the access policies on the respective data stream. The security punctuations are woven into the original data streams when access restrictions are to be supported. If at some point in time the GPS position stream is restricted to a certain subset of subjects (users) a corresponding security punctuation is generated and is woven into the output stream to tell subsequent operations about the access restriction setting which has changed. Security punctuations are implemented by two punctuation types: A *data security predicate* which controls access to data elements and a *query security predicate* which controls access to queries. To control data access two possible approaches are proposed. The first approach is a *security filter* approach which provides the use of the so-called *security shield plus* operator (SS+ operator). SS+ operators are integrated into the original query and filter data elements according to security punctuations. Here filtering for security punctuations is directly integrated within the query processing. The second strategy consists of *rewriting the query* and relying on existing operator implementations. To support the filtering of security punctuations the query predicates must be rewritten such that—beside

[2]Recall: A punctuation is a data element within the data stream which carries arbitrary information useful for the DSPS. It may carry information about the characteristics of the data stream. Punctuations have been introduced in Section 3.3.

the original predicate condition—the selection operator filters out data elements which do not meet the access restrictions defined by the security punctuations.

5.3.1 Discussion

The considered approaches propose interesting features and give valuable directions. However, the approaches are not suitable for NexusDS. The query rewriting to enforce the security policies is a major drawback since a complete rewrite functionality presupposes the query processor to know semantically all involved operators. NexusDS is open and extensible, thus allowing arbitrary operators. The centralized approach in Secure Borealis guarantees that security policies are enforced, but it is not feasible since it represents a potential bottleneck limiting the amount of queries that can run in parallel. Secure Borealis and ACStream in principle allow to integrate custom operators into the system. However, no precautions are taken to prevent uncontrolled outflow of data. Moreover, the data access granularity is not adjustable to domain-specific needs. Some data providers generally permit other subjects to use any of their objects (e. g. context data such as GPS positions). But eventually a subject may also want to restrict the access to and processing of the exact objects to an exclusive set of subjects, by at the dame time providing the data to others, only less accurate. Finally, the approaches presented only consider access control mechanisms. However, the way data is being processed is also of importance. E. g., certain data should not cross certain administrative boundaries and thus processing should be limited to a certain set of processing nodes.

Our augmentation approach is an extension on the SP graph level and shows some important advantages compared to the other approaches presented:

- The semantics of the operators need not be known in order to adapt the SP graph to meet the security restrictions.
- The operator model presented is flexible in the way it embeds the operator within a box. Thus also operators which are not designed to work with the security framework are still usable.
- Our approach allows to define and integrate arbitrary granularity filters to adapt the data details and to still allow processing them at the cost of some reduction in data quality.

In the following sections we present the architecture and the characteristics of our security framework as well as the augmentation technique that is implemented as part of NexusDS.

5.4 Security Control Framework

The security framework in NexusDS meets the security goals defined in Section 5.2.2. In this section, the security framework for security compliant processing of streamed data is presented. Therefore, first basic assumptions for the security framework are presented. Then, we go into detail on the functional capabilities.

5.4.1 Basic Assumptions

Each subject interacting with the DSPS must be assigned a unique identity. This means that for each user, operator, and NexusDS node there must be assigned a unique identity. A variety of solutions already exist, such as identification by a combination of *name* and *password*. We assume that subjects can identify themselves by a valid name and password combination. Furthermore, all services and operators communicate using an asymmetric key algorithm. So, data is encrypted with the public key of the receiver and only the receiver is able to read the data by using its private key. To ensure liability, it is important to track all actions a certain subject performs. Therefore, the corresponding log information must be stored in a restricted and fail-safe area.

Subjects are associated with a set of security policies which are managed by a policy management component. These policies define access and process conditions for the subjects and the affected objects. Policies are divided into Access Control (AC) policies, Process Control (PC) policies, and Granularity Control (GC) policies. Each policy type covers a certain aspect of security according to the protection goals and are detailed in the following.

First, we start presenting the different security control patterns. Afterwards, the underlying approach to augment the SP graphs with security policies is shown. This must be done before deploying the SP graph to ensure that all security policies are met.

5.4.2 Security Control Patterns

Access Control: AC policies ensure data integrity and data confidentiality in terms of hiding data that subjects are not allowed to access and provide a mechanism to trace actions of subjects accessing data. AC policies enforce an *all-or-nothing* semantic. Each AC policy is uniquely identified by a *policyID*. The operators' and services' provenance accessing the data is certified by their digital signature [94].

Process Control: PC policies ensure acceptance of the execution environments and limit the extent of visible data. PC policies apply to *nodes* thus defining a set of NexusDS nodes which are allowed to execute certain *operators*. Furthermore, it is important to limit the *extent* of data, i. e., the currently visible window of data, for the operators. In this way it is possible to control the quality of aggregates such as traces of mobile objects. PC policies influence the placement of operators as part of the SP graph deployment. In [44] a flexible operator placement strategy has been presented that allows to influence the actual placement according to constraints. Such a constraint might be e. g. to limit the execution to certain NexusDS nodes.

Granularity Control: GC policies basically define *filters* that apply to operator *slots*. A slot unambiguously defines an operator-related input or output. These filters might be applied *before* data is sent to subsequent operators or *after* data has been received by this operator. This basically depends on the concrete configuration of the DSPS.

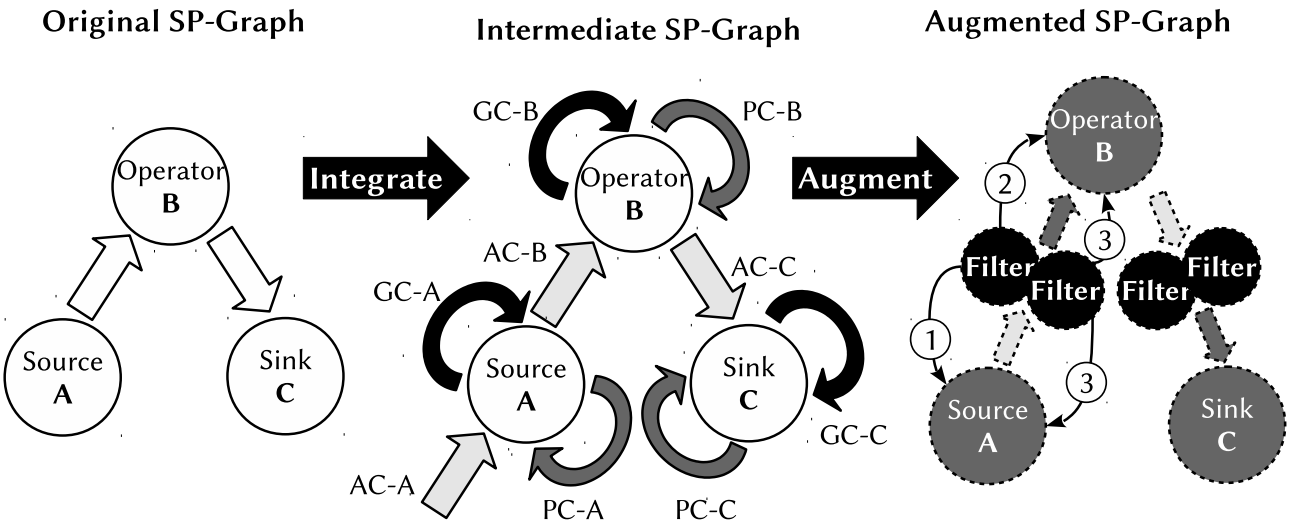


Figure 5.2: Augmentation principle of the secure framework. The original SP graph formulated by applications is translated into an equivalent one by integrating access control, process control, and granularity control patterns. In the second step the integrated security control patterns of the SP graph are translated into an augmented SP graph. Afterwards, the augmented SP graph is ready for deployment by an appropriate deployment algorithm.

5.4.3 Mode of Operation

The principal mode of operation to augment SP graphs is depicted in Figure 5.2. The starting point is the original SP graph shown in the left part of Figure 5.2. For each subject–object pairing there is a security policy defined which must be met. This step is denoted by the *integration process* which ends up in an intermediate SP graph (as shown in the middle of Figure 5.2). The intermediate SP graph has AC policies, PC policies, and GC policies attached to their operators. Each policy type is shown in a different shade of gray. Integration ensures that relevant security policies are integrated into the SP graph. The integration process consists of three steps: AC integration, PC integration, and finally GC integration.

In the AC integration phase, it is first checked for AC policies which define data access to the data involved into the computation task that are relevant to the subject running the application. This is denoted by **AC-A**, which defines whether the subject is allowed to access **Source A**. After that, the AC policies for **AC-B** (**Operator B**) and **AC-C** (**Sink C**) are attached to the SP graph. Note that for all subjects involved the AC policies must be considered in this phase, i. e. also for operators and NexusDS nodes.

The second phase, PC integration, consists of checking for PC policies relevant to the subject (e. g. a user) stating if the subject is allowed to execute the operators of the SP graph. Analogous to the AC integration the respective PC policies are attached to the SP graph. Besides the user-related PC policies there might also exist operator-related PC policies that limit the execution of a certain operator to a concrete set of NexusDS nodes. Thus, all existing PC policies for all subjects involved (including entities such as users, operators, or NexusDS nodes) must be attached to the SP graph.

The GC integration denotes the final phase before augmentation starts. Analogously to the previous integration phases, in this phase GC policies are attached to the SP graph. In Figure 5.2 these are displayed as **GC-A**, **GC-B**, and **GC-C**. The GC policies describe data transformation techniques to manipulate the original data so that the subjects involved only access the granularity of data they are allowed to. GC policies represent a refinement of the AC policies and PC policies. At this point the intermediate SP-graph consists of the original SP graph with security policy information for all subjects and objects attached to it.

After the integration process the *augmentation process* translates the intermediate SP graph to an augmented SP graph which is shown on the right side of Figure 5.2. First, the security policies must be checked for consistency before continuing, i. e. all AC policies and PC policies must be checked for consistency. This means that all senders must verify whether the attached receivers of their data are allowed to access the data depending on their attached PC policy. GC policies need not to be checked since they are refinements of the AC policies and PC policies to filter data. The AC policies map to interconnections between the involved operators which semantically means that a certain operator is allowed to access data from a previous one. The PC policies influence the AC policy interconnections since they influence the selection of NexusDS nodes the operator can be executed on. The GC policies map to filters which allow a fine-grained access to the single data streams. The placement of the filters may be done in three different ways. Referring to Figure 5.2, they may be placed on the sender side (1), on the receiver side (2) or on both sides (3). The actual placement depends on the receivers attached to a certain output stream. In Figure 5.2 this is shown for the combination **Source A** and **Operator B**. (1) is always preferred and selected to *limit the transferred data* volume. This is beneficial if the receiver is a mobile device and has stringent energy and bandwidth constraints. (2) is used if there is *more than one receiver* attached to the output stream of an operator's output slot having different filters. Thus, the filters are executed on the respective receivers. For (3) we have a *work sharing approach* which is selected if **Source A** and **Operator B** are both running on a mobile device or if the filtered output stream is used by multiple attached operators which need a dedicated filtering themselves.

5.5 Security Framework Insights

After the security pattern discussions we now present some internals about the characteristics of our security framework for security compliant processing of data streams as well as its implementation within NexusDS. Finally, we provide some details on the operator framework and its mode of operation.

5.5.1 Security Features of NexusDS

NexusDS is an open DSPS designed for processing context data streams with extensive capabilities for domain-specific adaptation and support for the protection goals described in Section

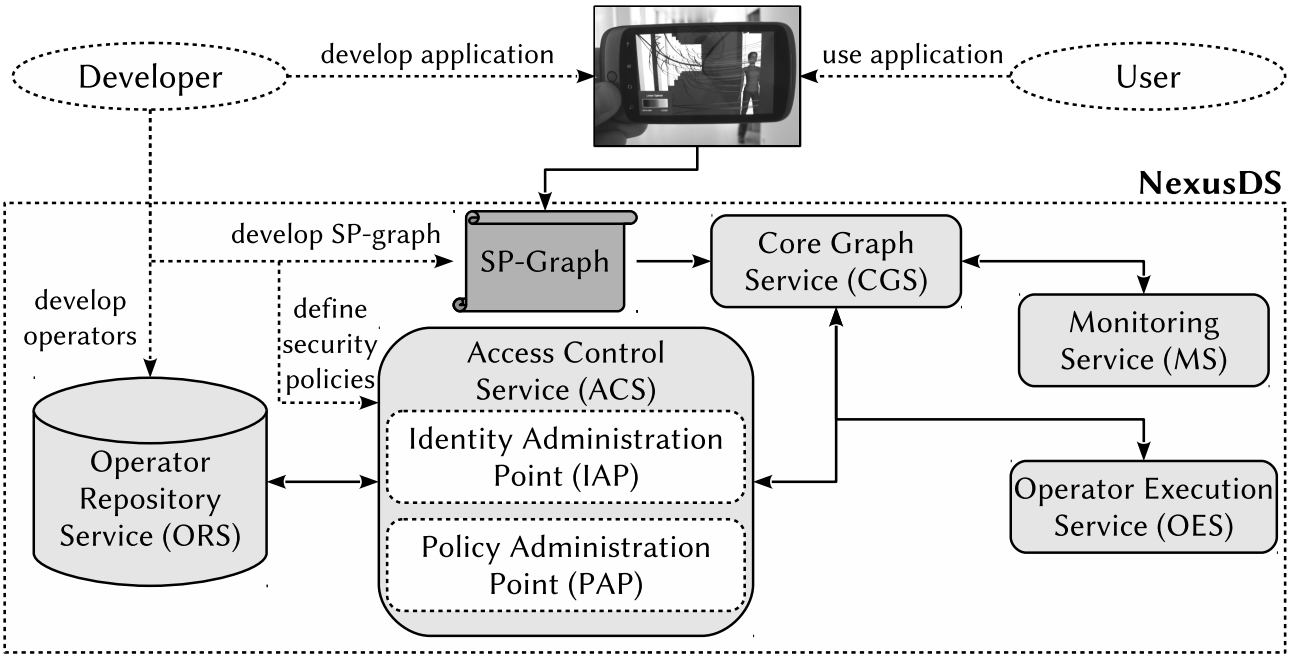


Figure 5.3: Simplified architecture of the targeted security concept.

5.2. Its architectural organization is depicted in Figure 5.3. NexusDS supports the distributed processing of streamed context data by execution of SP graphs on a heterogeneous network of NexusDS nodes. NexusDS allows flexible adaptation of the system functionality through the integration of customized services and operators. The SP graph can thereby be annotated by means of restrictions to influence the concrete deployment and execution of the SP graph. For example, deployment restrictions limiting the selection of a physical operator can be defined in this context. This restriction has an influence on the deployment process of the SP graph. Besides that, also runtime-specific restrictions can be defined. If an operator has specific parameters, these parameters can be adjusted to meet certain conditions. If we imagine a domain-specific rendering operator the resolution might be such a parameter.

The Core Graph Service performs the augmentation of the SP graph, by inserting missing filters according to the defined GC policies. This task is performed with the help of the Access Control Service, which was introduced in detail in Section 3.5.5. Beside others, the Access Control Service is composed of an Identity Administration Point and a Policy Administration Point. The Identity Administration Point is responsible for identifying all subjects and objects available within the system. The Identity Administration Point also provides the Public Key Infrastructure (PKI) to secure the communication of services and the data streams between operators against unauthorized access. Furthermore, certificates for operators executed in the secure environment are created with the PKI. Certificates are needed to validate if operators are known by the security framework and can be subsequently executed. All services and operators interacting with the secure environment need a corresponding key and must be certified via the Identity Administration Point. The Policy Administration Point holds the policies for accessing and processing data.

The fragmentation of the SP graph is done by our M-TOP approach, presented in Chapter 6. M-TOP is a multi target operator placement approach for heterogeneous environments. M-TOP considers annotations at SP graph level. These annotations in the original work from Cipriani et al. [44] focused on QoS aspects such as *"latency should not exceed a certain value"*. However, the basic concept also supports annotations such as the security policies, annotated at SP graph level to adapt deployment decisions. The single fragments are distributed across appropriate Operator Execution Service which are instances of NexusDS nodes running a certain execution environment for the operators of NexusDS. Each Operator Execution Service instance runs on a different NexusDS node. These services represent the central components of NexusDS to process data streams. The Monitoring Service collects runtime statistics for the NexusDS nodes running the operators and provides hints which Operator Execution Service instances to use for each fragment. These statistics are exploited to enhance future placement decisions.

5.5.2 Security Compliant Operator Framework

Besides the different architectural entities necessary for security policy management, the actual data processing facility must support the notion of security policies in order to make the security framework work. As introduced in Section 4.2, for that purpose we adopt a *black-box principle* decoupling the definition of processing logic (in terms of operators) and security policies. This facilitates the development of operators since developers can focus on the actual processing logic. Furthermore, already existing operators can be easily embedded and re-used without the need to write dedicated ones. To support security policies, also the corresponding AC policies, PC policies, and GC policies must be defined. The SP graph is augmented by the security policies valid for the subjects and objects involved in the SP graph definition.

To create an environment for safe operator execution (the same holds for source operators and sink operators) the operators are embedded within a *box*. The box provides the execution facilities for operators and includes *decoders*, *filters*, and *encoders*. Each of these entities is associated to an ingoing or outgoing slot. The operator itself is contained in the box and only receives data that has been altered complying to the security policies.

Figure 5.4 illustrates the embedding of an operator. First of all, the surrounding box, not the operator itself, is connected with all incoming data streams (1). The decoders decrypt all incoming data streams and signal to the encoders when to add the security punctuation to the outgoing streams (2). Then the box applies the necessary filters to the incoming data streams before they are transferred to the operator (3) and forwards the decrypted and filtered data to the actual operator performing the operations (4). When the operator has finished, the box receives the processed data from the operator and encrypts it through the encoders which also check if a punctuation must be inserted before the data element is forwarded (5). The time instant when the security punctuations must be inserted is given by internal time stamps, i. e. each data element is assigned an internal time stamp. Finally, the data is passed on

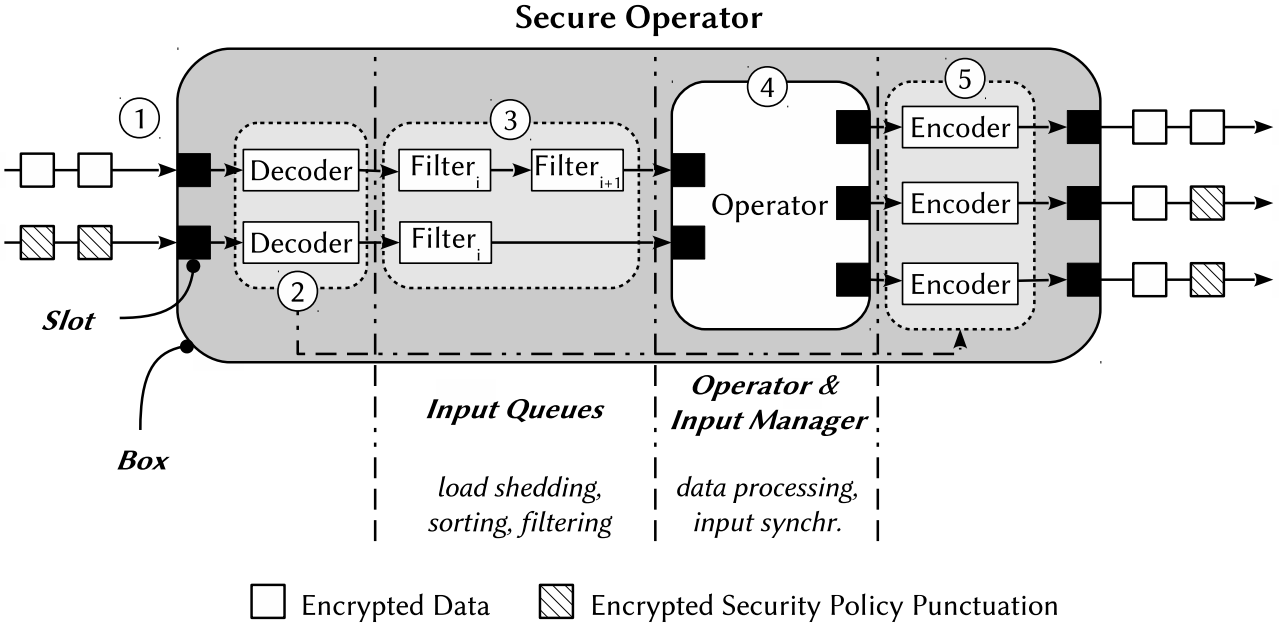


Figure 5.4: Secure operator which is part of the operator framework supporting security policies. Dashed arrows indicate control flow interaction with architectural components and solid arrows indicate data flows.

to subsequent operators. For sink operators, the figure looks similar, only that sink operators have no outputs.

The original operator model presented in Section 4.2 is presented in a slightly modified way to highlight security policy support. Thereby the box decodes the incoming data elements and the queue applies the filters on the data requested by the input manager. The input manager is not shown in Figure 5.4 for sake of simplicity. Also, after the operator has finished the box receives the processed data elements and encrypts them by calling an appropriate encoder.

Analogously to the operator, the source operator is depicted in Figure 5.5 and is also embedded in a box that carries out all security relevant operations. However, this does not show inputs, since the source produces data. An example for such a source is the GPS sensor from the example in Section 5.1. For source operators, after the data generation process (performed by the embedded *Source*) the registered filters (GC policies) must be applied to the respective data streams (1). As with the operators described beforehand, different filters might be defined for each outgoing data stream. After the filtering, the data must be encrypted and signed in order to prevent manipulation from a third-party (2). The encrypted data is forwarded to subsequent operators (3).

5.5.3 Security Characteristics

It is important to note that for encryption and decryption of the data streams a symmetric key approach is used. For each SP graph instantiated and executed a separate key is generated.

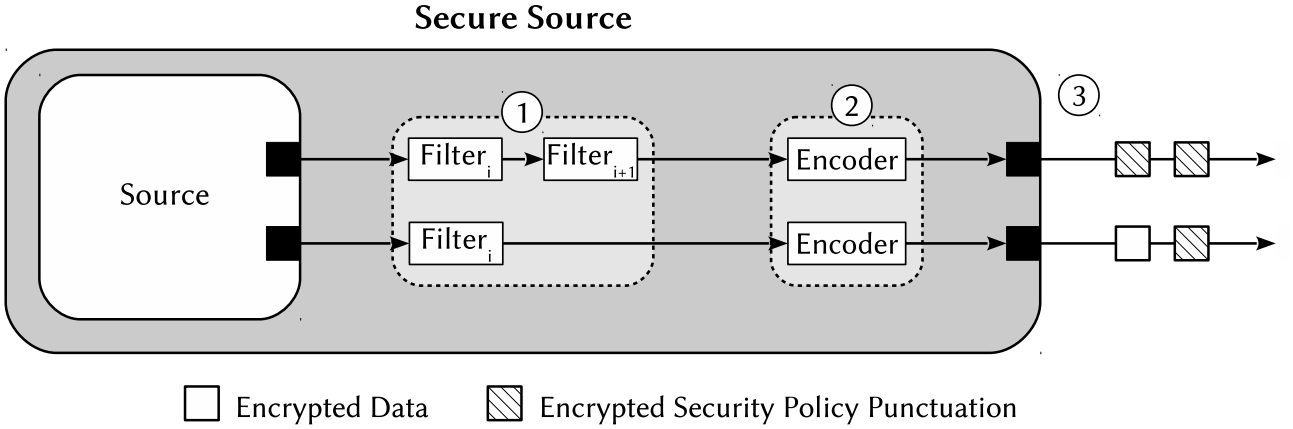


Figure 5.5: Secure source which is part of the operator framework supporting security policies. Solid arrows indicate the source's produced data streams.

This segregates single SP graph instances running, maybe, on the same machine. A time to live (TTL) period is assigned to each generated key. Before the TTL is reached, a new key is generated and propagated to the affected SP graph entities. This might result in a slightly higher overhead but increases security for long running queries. The concrete TTL assignment for the keys thus strongly depends on the runtime of SP graphs.

All operators (including particularly the source operators and sink operators) of which access should be controlled, are provided by the Operator Repository Service. The good behavior of an operator is verified by its associated certificate. Certificates are awarded by a separate certificate authority that attests through various checks (code check, verification or test) the respective subject to be safe. Thus, although no guarantee is given for good behavior, a useful degree of control is nevertheless carried out. The certificate contains a public and a private key (according to an asymmetric key approach). The subject the certificate belongs to is now able to create signatures to ensure its correct provenance. Therefore, a subject-related hash value is computed and encrypted with the private key. This signature is validated each time the operator is executed. Therefore, again the hash value is computed and encrypted. The result is compared to the existing signature. If they match, the execution can be carried out. Otherwise the subject has been manipulated at some point in time prohibiting the execution of this subject in the secure environment.

5.6 Deployment and Runtime

According to the mode of operation and the security control patterns discussed in the sections before, in this section the implementation of our security framework in the NexusDS system is presented. In detail, the augmentation process is presented by starting with an excerpt of the original SP graph document in XML notation, as shown in Listing 5.1. The example in Figure 5.1 is revived as lines **7 – 13** describe the GPS data source (**S1**).

```
1  <awml:awml xmlns:nsat="http://www.nexus.uni-stuttgart.de/1.0/NSAT"
2          xmlns:nsas="http://www.nexus.uni-stuttgart.de/1.0/NSAS"
3          xmlns:nscs="http://www.nexus.uni-stuttgart.de/1.0/NSCS"
4          xmlns:awml="http://www.nexus.uni-stuttgart.de/2.0/AWML">
5
6     <awml:nexusobject>
7        <eas:block>
8           <nsas:value>
9              <eas:blockType>source</eas:blockType>
10             <eas:blockID>ResultSetSource0</eas:blockID>
11             <eas:classURI>urn:java:de.uni_stuttgart.nexus.federation.streamFederation.sources.core.
                   resultSetSource.GPSSource</eas:classURI>
12          </nsas:value>
13       </eas:block>
14
15       [ ..... ]
16    </awml:nexusobject>
17    [ ..... ]
18 </awml:awml>
```

Listing 5.1: SP-graph excerpt for the source operator retrieving data from third-party servers.

The first part (lines **1 – 4**) consists of namespace definitions. The remaining part consists of different sections, defining the SP graph structure, including operators, links and so forth. The displayed listing is an excerpt of the operator definition section. The code defines a source operator named **ResultSetSource0** (also being an unique identifier for this source operator). The class representing this source operator is defined by the **eas:classURI** attribute. The remaining sections (denoted by **[...]**) are not shown for simplicity reasons.

5.6.1 Augmenting SP-graphs with Security Policies

Figure 5.6 illustrates the most important aspects of the augmentation process as implemented in NexusDS. The figure picks up the introductory example scenario from Figure 5.1. The three-staged augmentation of SP graphs by AC policies, PC policies, and GC policies is described in the following:

First, the AC policies are considered. Both Bob (**T1**) and Alice (**T2**) are allowed to access Mike's private data. Furthermore, also the *FF visualization* (**O2**) must be able to access the data of the previous combine step (**O1**), since operators represent a subject which needs access permission. Therefore, the operator-related AC policies attached to the SP graph are evaluated. Additionally, the certificates of all operators are verified if a signature is provided.

The PC policies influence the placement of operators for SP graph deployment. The set of nodes corresponding to the PC policies is determined by the operator itself via meta data.

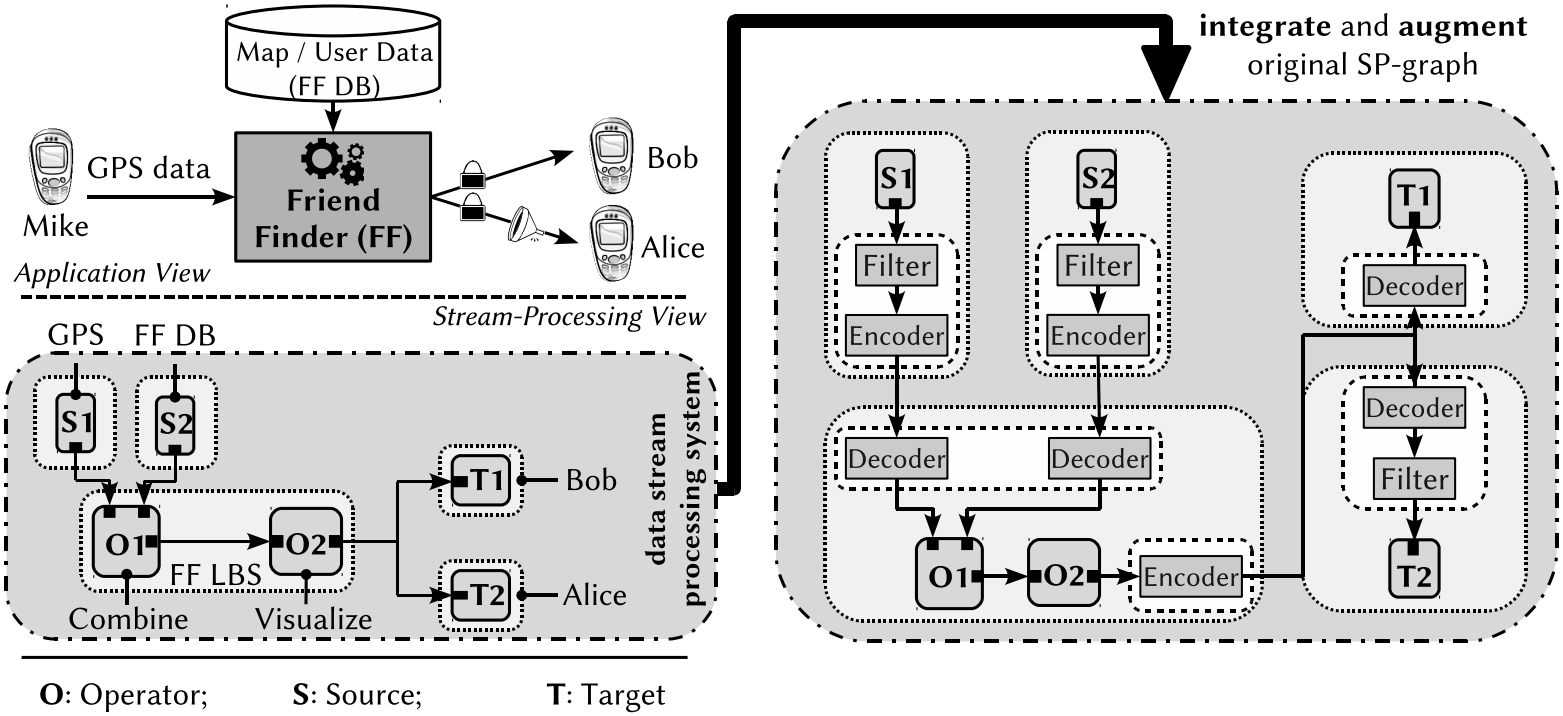


Figure 5.6: Illustration of the augmentation concept. The original SP graph on the left side is augmented by the corresponding measurements declared in the AC policies, PC policies, and GC policies respectively.

E. g. the visualization operator (**O2**) might need a GPU to run properly. Additionally, the PC policies define a second set of nodes that are needed by an operator in order to process its predecessors data. The intersection of these two sets constitute the set of nodes the operator can be executed on. Furthermore, the quality of aggregates can be controlled by PC policies by limiting the extent of visible data.

Finally, the SP graph is adapted according to the GC policies. The GC policies are evaluated for each operator and the corresponding filters are integrated into the SP graph, e. g. when Alice (**T2**) gets Mike's coarse location. Also appropriate encoders and decoders for data encryption are integrated into the SP graph.

At this point the augmentation phase is completed and the deployment phase starts. The deployment is presented in detail in the next chapter (Chapter 6). Each SP graph fragment is mapped to an Operator Execution Service instance which executes the contained operators, encoder, decoder, filters, and so forth. The augmented and deployed SP graph runs and generates data originating from the two source operators *A* and *B* until the sink operators *C* and *D* are reached.

Listing 5.2 is an augmentation of Listing 5.1 after the integration of the three security policy types. Two additional blocks are integrated into the SP graph document: **policies** and **filters**. In our example listing, for the source operator **ResultSetSource0** a policy is added. This source operator corresponds with the source operator *S1* from Figure 5.6, representing the source for the personal data on the mobile device. Each policy has a related operator, denoted by the **blockID** attribute. Furthermore, each policy also has an unique **policyID** attribute to uniquely identify the corresponding policy that applies here. The policy for the source operator *A* defines a filter. This filter applies to the output **slotID** 0 of the given **blockID**. **filterS** defines the signature of this filter to be sure the executed filter **filterURI** is the right one. The attribute **role** represents the user requesting the SP-graph execution. Finally, the attribute **policyID** correlates this filter to the policy defining it.

```
1  <awml:awml xmlns:nsat="http://www.nexus.uni-stuttgart.de/1.0/NSAT"
2          xmlns:nsas="http://www.nexus.uni-stuttgart.de/1.0/NSAS"
3          xmlns:nscs="http://www.nexus.uni-stuttgart.de/1.0/NSCS"
4          xmlns:awml="http://www.nexus.uni-stuttgart.de/2.0/AWML">
5    <awml:nexusobject>
6      <eas:block>
7        <nsas:value>
8          <eas:blockType>source</eas:blockType>
9          <eas:blockID>ResultSetSource0</eas:blockID>
10         <eas:classURI>urn:java:de.uni_stuttgart.nexus.federation.streamFederation.sources.core.
               resultSetSource.GPSSource</eas:classURI>
11       </nsas:value>
12     </eas:block>
13
14     <eas:policy>
```

```
15          <nsas:value>
16            <eas:blockID>ResultSetSource0</eas:blockID>
17            <eas:policyID>aff337fe−abcf−4077−bb3c−743f4562b424</eas:policyID>
18          </nsas:value>
19        </eas:policy>
20
21        <eas:filter>
22          <nsas:value>
23            <eas:blockID>ResultSetSource0</eas:blockID>
24            <eas:slotID>0</eas:slotID>
25            <eas:filterS>kqiR+IjnnRwui4J3zG1hRmxQj [...] Qwa0Pv02gICiJ382Pw</eas:filterS>
26            <eas:filterURI>urn:java:de.uni_stuttgart.nexus.federation.streamFederation.filters.secure.
                  resultSet.ResultSetFilter</eas:filterURI>
27            <eas:role>poweruser</eas:role>
28            <eas:policyID>aff337fe−abcf−4077−bb3c−743f4562b424</eas:policyID>
29          </nsas:value>
30        </eas:filter>
31        [ ..... ]
32      </awml:nexusobject>
33      [ ..... ]
34    </awml:awml>
```

Listing 5.2: SP-graph excerpt for the source operator retrieving data from third-party servers.

5.7 Reacting to Security Pattern Changes

For security policy changes an attribute called **immediate** is evaluated to adapt the change propagation into the system. If **immediate** is set to **true**, the changes are applied immediately, independent of the data generation time. This in turn means that the policy changes are immediately sent to all Operator Execution Service instances executing an affected fragment, and the changes are forced. If **immediate** is set to **false**, changes are not applied immediately but deferred by exploiting the security punctuation mechanism. The security punctuations are then transported together with the actual data elements.

The differentiation of security change propagation can be exploited by DSPS providers to create different accounting profiles. E. g., an immediate application implies a certain overhead which in turn means that the subject which changed the security policy and to which the policy belongs to has to pay an additional fee. The changed security policies are propagated by the channels the data is transmitted with. The overhead introduced by this technique is relatively low compared to the one described before which results in a lower additional fee to be payed by the subject.

If security change propagation is deferred, changes are propagated by weaving punctuations into the affected data streams. The punctuations are transported automatically to the correct

operator instances since they take the same route as the actual data. These punctuations are evaluated by the security framework the operator is running in and applied accordingly.

Various modes of security policy adaptations are possible. Based on the augmentation and deployment approach, a policy change can be realized at a SP graph level. Possible modes for security policy adaptations are:

- **Offline**: This mode requires the executed SP graph to be stopped in order to make adaptations according to security policy changes. Once the SP graph is stopped, the necessary adjustments are made and the entire deployment step is re-executed. The buffered data by the single operators is lost and all computation starts again from the beginning.
- **Shadow**: This mode creates a new SP graph instance with modifications resulting from the security policy changes. The new SP graph is deployed and execution of the old one is shortly interrupted. The data channels are redirected and the data buffered by operators is transferred from the old SP graph to the new one.
- **Online**: This mode allows changes without stopping and restarting the current SP graph. It is important that the DSPS allows modifications to SP graphs being executed (e. g. insert an additional filter or remove one) and that the DSPS supports a migration concept for operators to shift operator execution to another node, preventing an interruption or complete abortion of processing.

The different adaptation modes strongly depend on the kind of security policy changes and the capabilities of the DSPS. NexusDS currently supports the offline mode.

5.8 Summary

With the rapidly increasing number of mobile devices equipped with GPS sensors and mobile Internet connections, the use of data stream processing is increasing in many application areas. This chapter has presented a security framework and its integration into NexusDS . The security framework deals with the requirements of modern applications relying on the data stream processing paradigm. Thereby, the security framework proposes different security control patterns, i. e. AC policies, PC policies, and GC policies, which can be assigned to different system entities. The defined security control patterns are exploited to ensure a safe processing of sensible data. This is achieved by augmenting SP graphs with the corresponding AC policies, PC policies, and GC policies. By the proposed security framework it is possible to adjust the density of information that is going to be processed as well as to limit access to data. The following chapter highlights M-TOP. M-TOP performs the deployment of SP graphs by exploiting the respective annotations.

6 Stream Processing Graph Deployment

In the past decade, DSPSs gained great attention. They are well suited to address the challenges in processing high-volume and real-time data. In particular, DSPSs such as [2, 40, 57, 78] have been in the focus due to their inherent ability to distribute load among different participants. This is a key feature for scalability, as it avoids bottlenecks when processing potentially unbound data streams. In general, these systems provide a declarative query language and support the continuous distributed processing of incoming data streams. In some DSPSs, the application developer creates SP graphs directly. A topic of interest is how to distribute a SP graph across different computing nodes w.r.t. certain objectives, e. g. to avoid single points of failure or guarantee a certain Quality of Service (QoS). This is important as the initial distribution has a big impact on the run time behavior of a DSPS. An inappropriate initial SP graph distribution degrades execution and may lead to big overhead during run time, e. g. by migrating operators with heavy state.

In this chapter, the multi-target operator placement (M-TOP) of SP graphs is presented. M-TOP is a QoS-aware multi-target operator placement algorithm. M-TOP applies to the operator placement problem in data stream processing environments and considers a set of application-specific QoS targets for operator placement. Provided there is a set of nodes to place a set of operators, M-TOP searches for an operator placement that fulfills the specified QoS targets. The M-TOP heuristics thereby aim at eliminating placements that do not lead to suitable solutions.

For the operator placement process different approaches are feasible. We assume application developers to create corresponding SP graphs. Our hands-on experience has shown that developers are interested in explicitly providing the QoS targets to influence directly the SP graph distribution and so the operator placement process. By this, the application QoS requirements are best reflected to the process of operator placement in DSPSs.

Sections 6.1 and 6.2 provide a problem description for the operator placement in a distributed environment. Then, in Section 6.3, related work is presented and discussed. Section 6.4 intro-

duces the multi-target operator placement problem which is the focus of M-TOP and is presented in detail in Section 6.5. The sections 6.6 and 6.7 present details of the mapping step of M-TOP. In Section 6.8 we discuss the experimental evaluations performed to show the effectiveness of M-TOP. Finally, in Section 6.10, this chapter is concludes by a short summary.

6.1 Problem Description

As argued in Section 3.2 there is an adaptation problem in DSPSs which makes it difficult for applications—relying on the same processing paradigm, i. e. push-based stream processing—to integrate custom functionality seamlessly and moreover to influence the deployment process. Tight application integration in DSPSs is beneficial because data processing is tightened and the actual processing happens close the actual data. This improves data processing and allows reuse of already existing components, which avoids isolated solutions and redundancy.

The application of an interactive visualization application as the one presented in Section 2.3.1 visualizes the result of a SP graph operating on data streams and exploiting the functionality of DSPSs. The DSPS topology is utterly heterogeneous with devices ranging from desktop computers to mobile devices. The SP graph abrasively consists of three processing steps: *filter*, *map* and *render*. The filter operator is reusable by existing DSPSs, whereas the map and render operators implement custom logic and need to be implemented by the specific application developer. However, in contrast to the custom map operator, the render operator has additional specific requirements in order to run properly, e. g. a GPU. Hence, operators being deployed might have inherent restrictions that must be considered for operator placement when selecting devices going to execute the specific operator.

Beside the inherent restrictions, operators might also have restrictions which are application-specific (defined by their respective developers) or even user-specific. Assuming customers are classified in (a) with *gold*, (b) with *silver*, and (c) with *bronze* status[1], each one has its own priorities when executing the application:

- Customer (a) wants the bandwidth utilization maximized and requests a minimum of **10 MBit/s** since no additional accounting will occur. Furthermore latency must be minimized but should not be more than **500 ms**.
- Customer (b) wants the bandwidth utilization maximize requesting a minimum of **1 MBit/s**. Latency should be minimized not exceeding **2 s**. However, costs should be minimized not exceeding **1 cost unit**.
- Customer (c) wants just to maximize bandwidth utilization but requests a minimum of **500 kBit/s**, his applications needs to run properly. Utilization costs should not occur at all.

[1]Here gold stands for *all inclusive*, silver for *100% accounting on QoS utilization exceeding a certain threshold otherwise 50% accounting*, and bronze for *100% accounting on QoS utilization.*

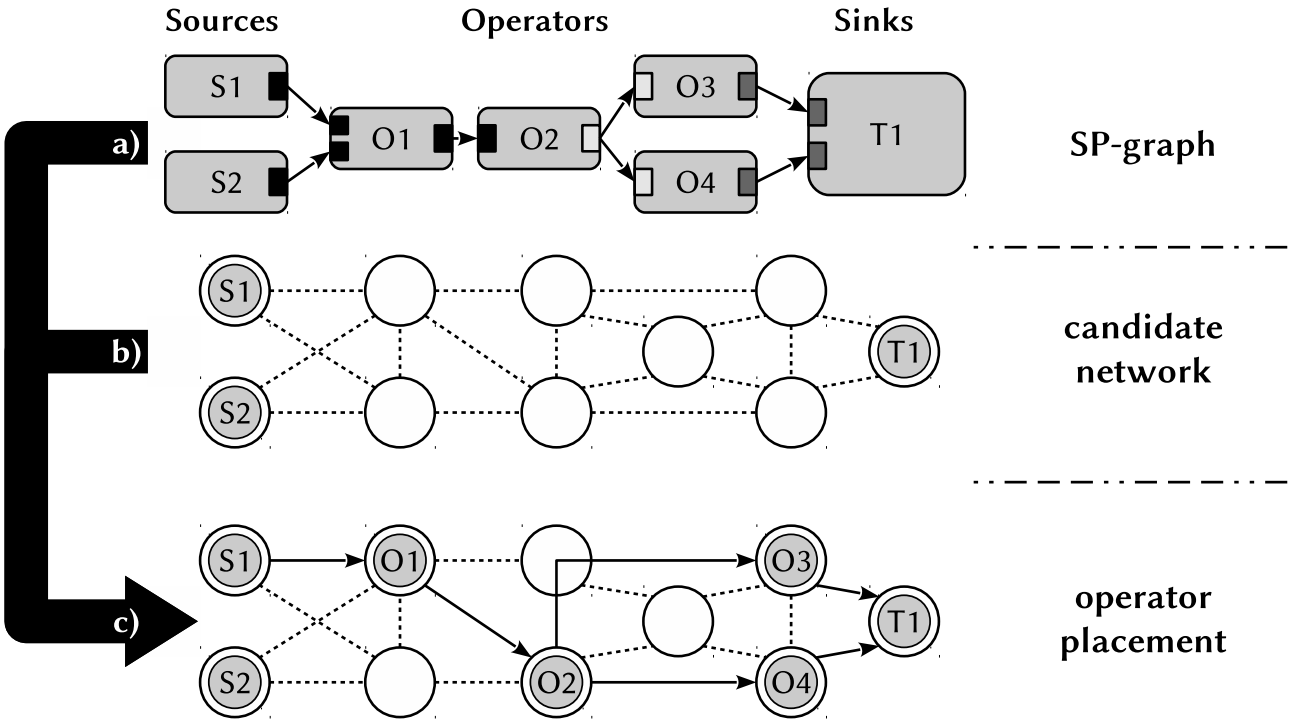


Figure 6.1: Operator placement problem in DSPS.

For the simple scenario sketched, considering three objectives, the single customers (and accordingly the corresponding applications) ask for different QoS targets. As can be seen by this simple examples, target specifications might even contradict each other, e. g. customer (c) asks for a bandwidth maximization but for minimization of utilization costs. Hence, we need a flexible mechanism to determine a suitable application-aware query graph distribution to meet the specified requirements which might be controversial.

6.2 Operator Placement Problem

We target the operator placement problem as depicted in Figure 6.1. Assuming all logical optimizations are done, placement of the operators of a SP graph can start, i. e. perform the so-called physical optimization. According to Figure 6.1, **a)** denotes the (logical) SP graph, **b)** the network of NexusDS nodes, and **c)** a possible operator placement of the SP graph on the network of NexusDS nodes. Parts of the query SP graph are *pinned* to certain NexusDS nodes. In our scenario from Figure 6.1, the source operators **S1** and **S2** as well as the sink operator **T1** are pinned to their respective data producing and data consuming NexusDS nodes. To deploy the SP graph **a)** in the network **b)**, a valid mapping for the (remaining) *unpinned* operators **O1** to **O4** is required.

The execution of such a query graph in distributed stream processing systems is deeply affected by

(1) the initial query graph distribution strategy that a certain system employs and

(2) the employed techniques to adapt the query graph execution at run time.

Literature (see Section 6.3) refers to the overall problem (comprising (1) and (2)) as the *operator placement problem*. In the context of this chapter, we refer to (1) as *pre-deployment operator placement* and to (2) as *post-deployment operator placement*. The pre-deployment operator placement finds a possibly stable operator placement for a given SP graph. So, the goal is to select suitable NexusDS nodes that will host operators and satisfy a predefined utility function. However, such DSPSs are usually subject to changes, thus post-deployment operator placement becomes necessary. These techniques are applied during SP graph execution time due to operating condition changes.

M-TOP addresses the pre-deployment issue of finding an operator placement for a given SP graph without considering postponed post-deployment adaptation techniques. The initial placement decision is important for efficient execution as it may avoid postponed migration of operators. Existing post-deployment techniques might be applied without loss of generality during query execution to adapt to changing load and infrastructure conditions.

6.3 Related Work and Classification

As operator placement is a topic of interest in research, many alternative placement strategies have been proposed. In Figure 6.2, a classification of the major approaches is provided. As depicted in this figure, the placement strategies differ in their objective: **single target** or **multi target**. Target is used as a synonym for objective. In **centralized** data stream processing system operator placement is not necessary. Multi-objective processing has also been targeted beforehand for DBMSs [20]. Contrarily, we focus on DSPSs where data is processed in a **distributed** manner and furthermore is typically done for a long-running time.

Single target techniques include finding an operator placement to reduce *system load* [149], *latency* [82, 114], or *bandwidth utilization* [6]. Approaches that aim at distributing operators according to *operator importance* as provided by the inquirer [10] exist. However, scenarios described in Section 6.1 cannot be implemented by a placement strategy presented so far, since we need a way to evaluate multiple targets while taking into account application requirements.

With **multi target** placement strategies, targets are either predefined and **fixed** or **variable**. Fixed means that targets are predefined by the placement strategy, e. g. bandwidth and latency. Contrarily variable allows to combine available targets arbitrarily, i. e. select bandwidth, latency or a combination of both.

Fixed and variable target definition can be further classified by **a priori** and **a posteriori** decision making techniques. A posteriori decision making means that a decision making entity, e. g. a user or a developer, select the solution out of many solutions computed that fits best his requirements. In contrast to this, a priori decision making defines techniques which support prioritized targets and thus calculates a solution that fits the required target prioritiza-

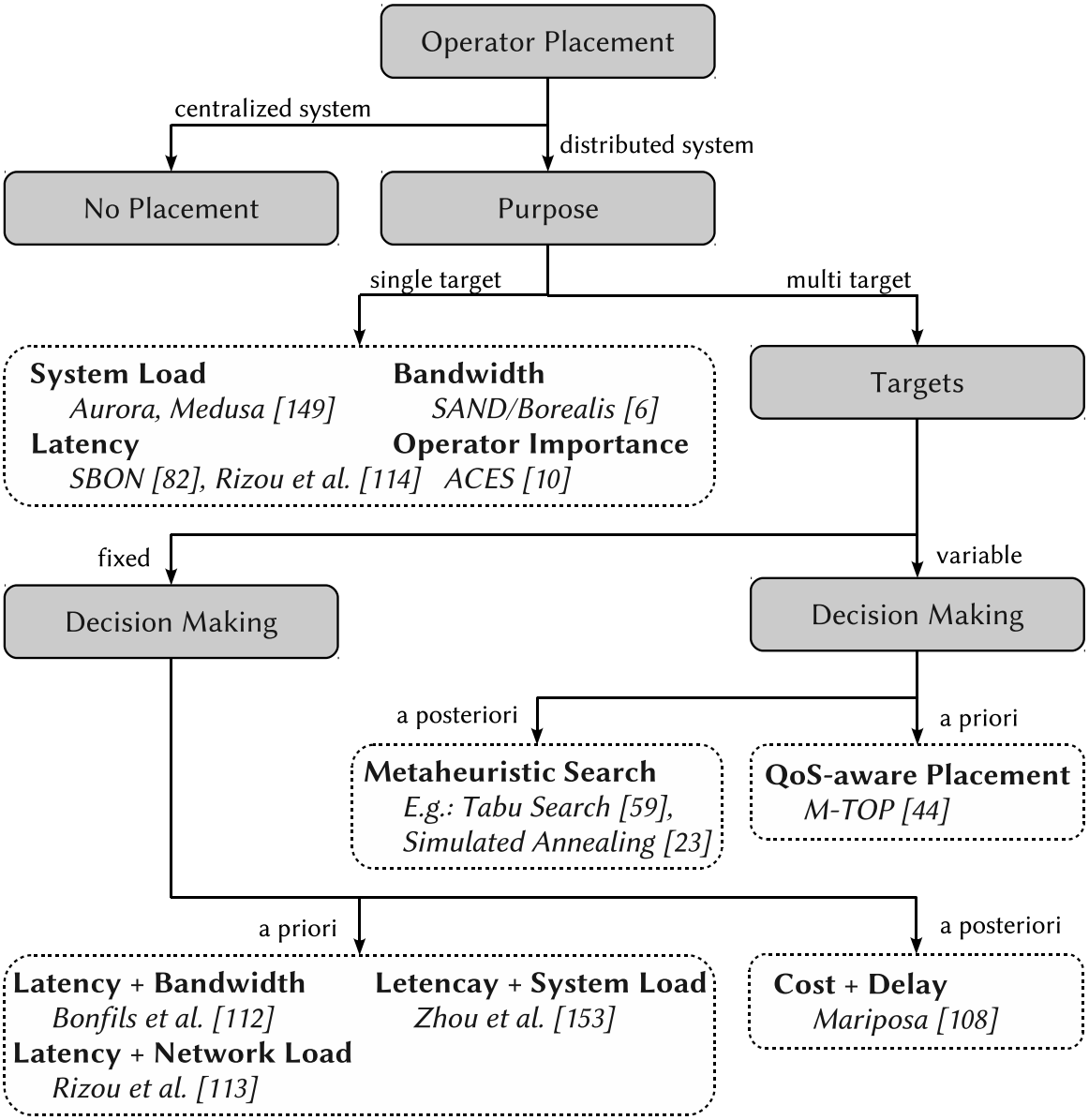


Figure 6.2: Classification of existing operator placement strategies.

tion. Fixed multi-target placement strategies with a priori decision making include minimization of *latency and bandwidth* [112], *latency and system load* [153], and *latency and network load* [113]. For this class of placement strategies a single solution is determined. Fixed multi-target placement strategies with a posteriori decision making is implemented by e. g. the Mariposa approach [108]. This approach tries to find non-dominated solutions for a given query and presents them to the user who has to choose the solution he prefers. These approaches do not consider variable application specific QoS constraints.

Variable multi-target strategies might also have a priori or a posteriori decision making. For a posteriori methods an arbitrary meta search method such as tabu search [59], simulated annealing [23], or some other heuristic can be used generating a pareto optimal set of potentially optimal solutions. However, a posteriori methods are computationally expensive, might be

long running and usually generate a set of possible solutions in which each solution is as good as another one in that set. It is difficult for a user or an application developer to choose the solution that will presumably best fit the application's requirements. If decision making is done a priori, operator placement can be performed by the M-TOP approach [44] proposed in this chapter, without further interaction.

Due to different design characteristics, each approach has its own strengths and shortcomings [81]. However, the approaches proposed so far do not satisfy the manifold requirements raised by the applications. For non-trivial applications[2] such as the visualization application described in Section 2.3.1, the placement problem goes beyond the state-of-the-art and raises new challenges. Therefore, a multi-target operator placement approach which allows to define specific targets of interest in order to account for application-specific requirements is needed, i. e. making the deployment process adaptable to application-specific needs.

To the best of our knowledge no placement strategy considers a priori specified multiple targets and thus allows to specify application-specific requirements by adding knowledge to the operator placement process. This, however, is necessary to integrate dedicated stream-based applications adequately and enhance placement decisions. Since many applications exist that also rely on the push-based stream-processing paradigm, it is a logical consequence to provide a specialized system with an appropriate operator placement mechanism to allow them tailoring the operator placement process.

The M-TOP approach is a QoS-aware way to influence the actual operator placement process by supporting QoS targets that met the application's needs. Our experience shows that application developers know the requirements an application has and specify QoS targets accordingly.

6.4 Multi-Target Operator Placement Problem

Multi-target operator placement is defined as the process of finding an operator placement for a given SP graph by taking into account multiple targets for operator placement. In other words, the task is to find a suitable sequence of computing nodes that maximizes or minimizes a certain set of QoS targets. We assume the QoS targets are known before operator placement. As already shown in Section 6.1, the QoS targets might be in conflict with each other. Generally speaking, a solution that simultaneously satisfies all targets in the best possible way does not exist. Moreover, usually more than one solution to this problem exists and we have to pick one that promises the best result—or at least one that is not inferior to others. Therefore, either an entity that picks up the solution depending on the personal preferences or the context a user might have is needed. Alternatively, the user or the application developer must select a solution finally.

[2] Non-trivial applications are those which QoS targets do not unconditionally fit into operator placement approaches commonly used in DSPSs, or to alternative applications running on the same DSPS. These applications generally originate from different application domains, each with their own specific requirements.

Our hands on experience has shown that users utilizing a DSPS and knowing the domain of interest usually know the targets of relevance to them. As presented in Section 6.1, multiple targets naturally arise as different applications might have different requirements and different contexts. Thus, the question here is how to select the right combination of computing nodes for each operator to fit the QoS targets specification provided by applications context best. More formally, the multi-target operator placement problem in our context is defined as follows:

Multi-target Operator Placement: Given a directed SP graph $Q_G = (O, E)$ where $o_i \in O$ is an operator and $E_i \subset E$ represents the set of edges from o_i to subsequent operators o_j ($E_i \cap E_j = \emptyset$, $1 \leq i < j \leq |O|$, $|O| \geq 2$). Furthermore a set of QoS-targets Q where $q \in Q$ defines a QoS-target and a set of computing nodes N with $n \in N$ representing a computing node capable of executing an operator o_i is given.

For a given SP graph Q_G the problem is to find a mapping of operators O to processing nodes N by considering the existing edges E and satisfying the specified QoS targets Q in the best possible way.

6.5 The M-TOP Approach

The M-TOP approach represents a top-k [20] retrieval mechanism for operator placement in heterogeneous environments. As M-TOP returns one solution k is equal to **1**. Therefore, M-TOP calculates the utility value for a given scenario, uses a combination of worst-fit and best-fit strategy and returns a solution. In the context of this thesis, QoS targets are defined as objectives with their respective *bottleneck conditions*, *relative importance factors* (trade-offs specifications), and *rank schemes*. To explain the QoS targets consider the following example:

QoS-target	bottleneck condition	relative weight	rank scheme
latency	300 ms	0.3	MIN
reliability	75%	0.7	MAX

The bottleneck condition specifies the worst value allowed for this target. E. g. for the QoS target *latency* the resulting computation value must not be greater than *300 ms*. The relative importance factors sets this QoS target in relation to others. E. g. *reliability* is more important than *latency* denoted by *0.3* and *0.7* respectively. The sum of all relative importance factors must be equal to 1. The rank scheme specifies if the QoS target should be minimized or maximized. E. g. the QoS target latency should be obviously minimized, whereas reliability should be maximized, denoted by *MIN* and *MAX* respectively.

As depicted in Figure 6.3, the M-TOP approach consists of six steps: *Conflation*, *Early Prune*, *Graph Assembly*, *Ranking*, *Mapping*, and *Execution*. *Conflation* consolidates operators in the

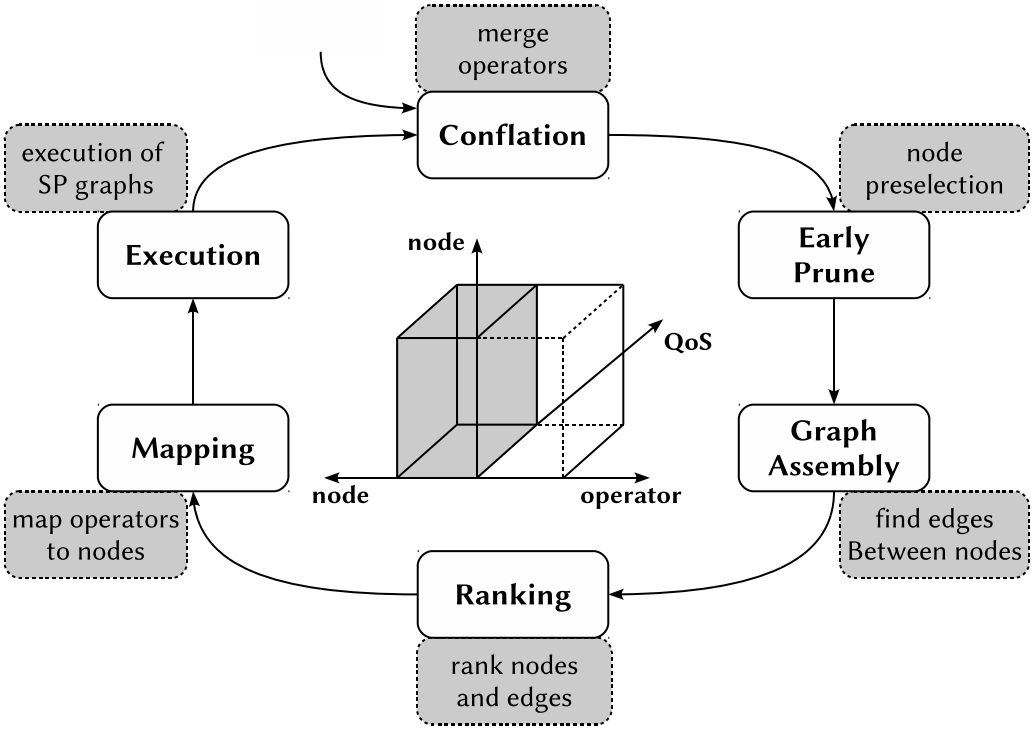


Figure 6.3: M-TOP approach — Conflation and Early Prune.

SP graph that should be deployed on a single node resulting in *virtual operators*. A virtual operator is a combination of the respective characteristics of the original operators it is composed of. Then, *Early Prune* determines possible candidate nodes for operator execution. These candidate nodes satisfy given constraints by their respective operators. Next, *Graph Assembly* creates a virtual overlay which describes the possible links between the single candidate nodes. The candidate nodes are ranked in *Ranking* to estimate the quality of each candidate node. *Mapping* finally performs the actual distribution by searching for a suitable mapping of operators to candidate nodes.

After the deployment the SP graph is executed in the *Execution* phase. During SP graph execution, statistics on the performance of the single operators and candidate nodes are collected. These statistics are necessary for operator placement decisions.

6.5.1 Runtime Statistics

Statistics are collected for *node-node* and *operator-node* combinations. The *node-node* statistics describe the QoS target based performance for the link between two nodes in the network. Analogously, *operator-node* statistics describe how an operator which is parametrized in a certain manner, performs on a NexusDS node regarding the considered QoS target.

Statistics can be imagined as a cube, as shown in the center of Figure 6.3. The cube is divided in two sections, *node-node* in gray and *operator-node* in white. The respective statistical data is collected and stored in the Monitoring Service, the storage backend for runtime measurements.

The statistical data consists of a fixed length time series of the corresponding QoS target representing a characteristic diagram and the probabilities that are expected for a certain value as well as minimum, maximum, and average values. A lookup returns minimum, average, and maximum values as well as the characteristic diagram for a specific QoS target.

These statistics are exploited to calculate the bottleneck condition intervals of the QoS targets specified to avoid greater utilization costs than those specified by QoS specifications. Thereby the bottleneck condition provided by the inquirer (representing either a lower or upper bound depending on the respective rank scheme) is enriched by the missing bound value to define the interval the candidates might be picked out. The service performs this task to manage the available resources. If such a QoS target is missing, the resulting interval has one infinite endpoint.

6.5.2 Conflation

As depicted in Figure 6.4, the first step *Conflation* combines two adjacent operators to a virtual one. Doing so, M-TOP adopts a *worst-fit heuristic* to distribute operators. Worst-fit means, that operators are uniformly distributed over the available NexusDS nodes. M-TOP assumes less operator migration tasks because overload situations on nodes executing the operators are avoided. As circumstances demand, conflating two adjacent operators to a *virtual* one, which results in a *best-fit heuristic* for the specified set of operators. Best-fit means that as many as possible operators are placed in the same NexusDS node, however without overloading it. Operators which are conflated and thus executed on the same node are marked as conflated operators, as depicted in Figure 6.4. Operator **Oa** and operator **Ob** are conflated to a single operator **O3**. Conflation combines operator requirements of both operators to a new virtual operator requirement. *Conflation* thus reduces the number of operator mappings to find in the *Mapping* step.

6.5.3 Early Prune

Early Prune determines candidate nodes for operator deployment and execution. For this purpose, as depicted in Figure 6.4, the attached operator requirements are evaluated as presented in Section 4.2.1.2. The requirements are matched against the available nodes capabilities to find suitable candidate nodes for the subsequent steps. Furthermore, the candidate node resources are checked for sufficiency to execute a certain operator. Collected operator statistics as well as live node statistics are considered, including measurements for CPU usage, memory usage, or disk usage. The nodes which properties match the operator requirements are attached to the corresponding list of compatible candidate nodes. This *candidate node list* is constructed for and attached to each operator for the subsequent steps.

Due to the utterly heterogeneous system topology this step helps to reduce the search space by pruning non-compatible candidate nodes. *Early Prune* also ensures that for each operator only compatible candidate nodes are selectable.

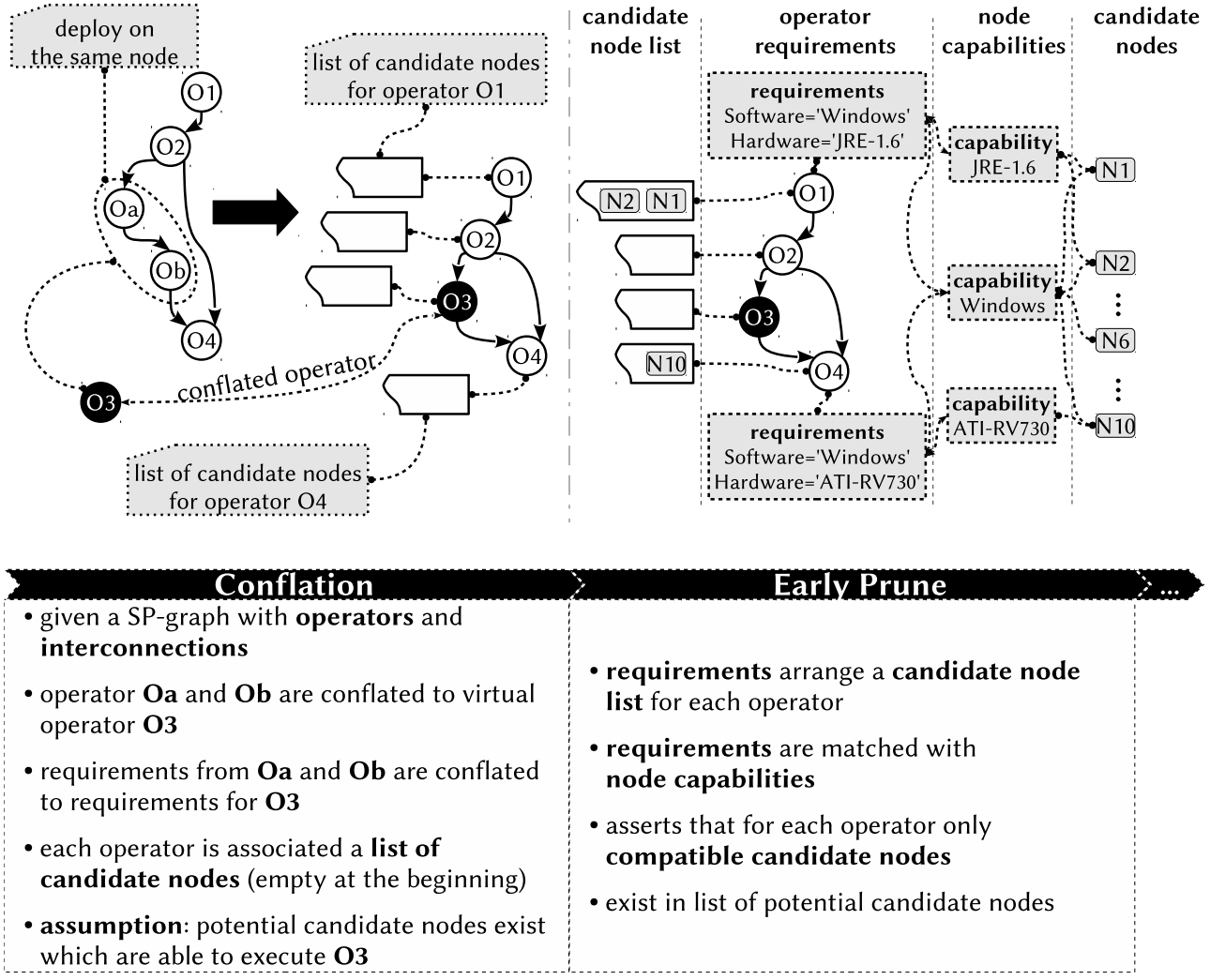


Figure 6.4: M-TOP approach — Conflation and Early Prune.

6.5.4 Graph Assembly

After completing the list of candidate nodes for each operator (virtual or original), *Graph Assembly* assembles a network of candidate nodes for the subsequent steps. The candidate node lists are aligned into *S-labeled* slices as illustrated in Figure 6.5. Two S-labeled slices are defined being adjacent if the related operators are interconnected. Each S-labeled slice represents a list of candidate nodes which relate to a certain operator, e. g. **S2** is the candidate node slice related to operator **O1**. It is important to note that in our sample scenario depicted in Figure 6.5, the slices **S0** and **S1** contain the source operators (which are pinned to the corresponding source nodes) and the slice **S6** contains the sink operator (representing the client application). Analogously, *E-labeled* slices represent a list of candidate links connecting candidate nodes. E-labeled slices contain the candidate links between the candidate nodes of two adjacent S-labeled slices.

Graph Assembly filters the candidate node and candidate links and removes candidates whose statistical values do not satisfy the *bottleneck conditions*. Therefore the average sta-

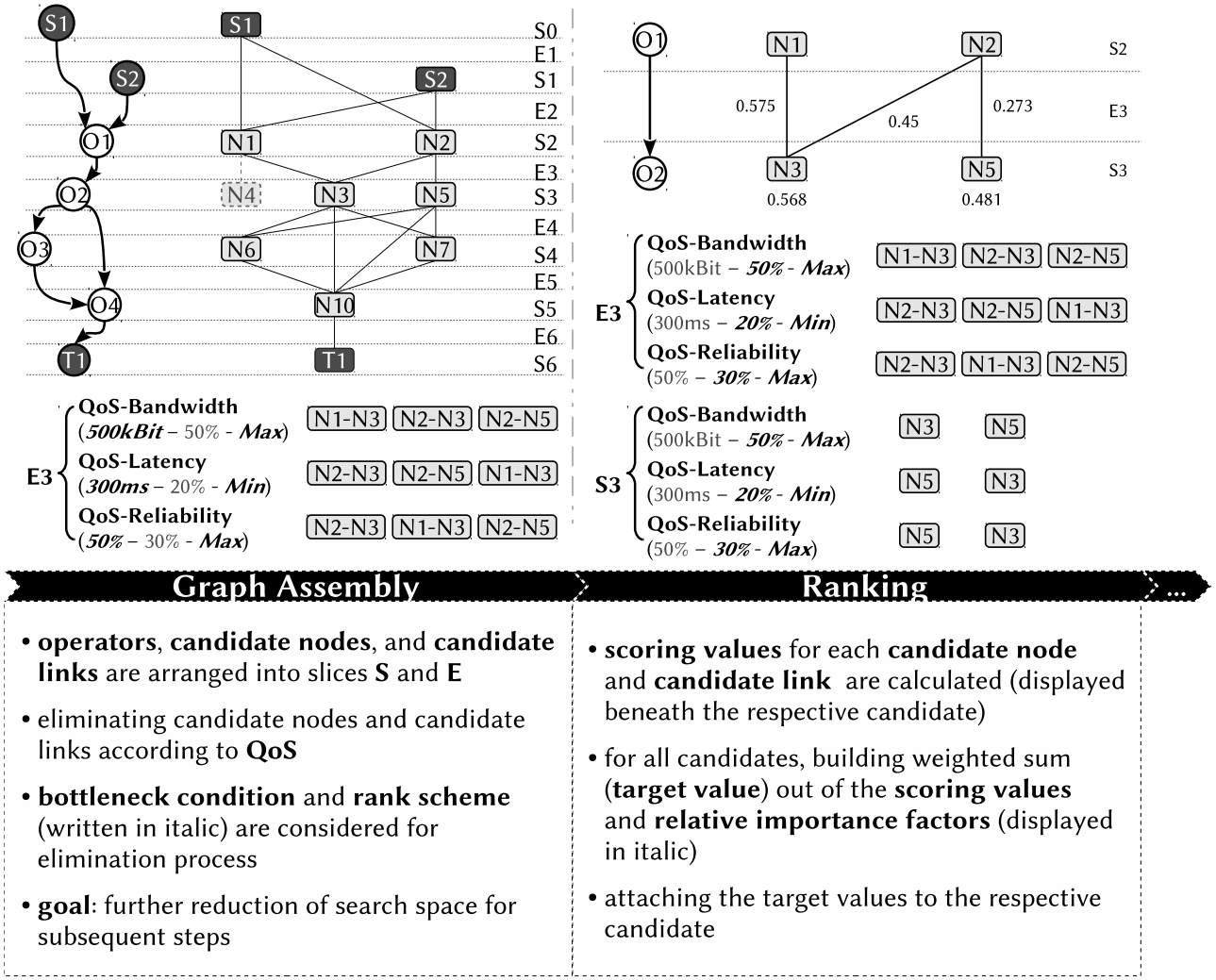


Figure 6.5: M-TOP approach — Graph Assembly and Ranking.

tistical values are used, as candidates are supposed to continue to show the same behavior as they have done in the past on an average. Thereby, for candidate nodes the *operator–node* statistics are considered. For candidate links the *node–node* statistics are taken.

In the example depicted in Figure 6.5, **300ms** is a bottleneck condition of the QoS *latency*. In this regard, the NexusDS node N4 is eliminated (shown as dashed box) from **S3** as this node does not meet the bottleneck condition specified. The interpretation of the bottleneck condition depends on the *rank scheme* provided. If the rank scheme is to minimize the QoS target, i. e., **Min**, the bottleneck condition constitutes an upper bound. The statistical value of a candidate node or candidate link must not exceed this value, otherwise it is removed from the respective slice. If the rank scheme is **Max**, the statistical value must not fall below this value. The relative importance factors are relevant for the next M-TOP step, i. e. Ranking.

The result of *Graph Assembly* is a candidate network that consists of candidate nodes and candidate links matching the specified bottleneck conditions and rank schemes. Like *Early Prune*, *Graph Assembly* also reduces the search space by removing candidate nodes and can-

didate links that do not match the specific QoS targets. Both steps reduce the number of candidates for each operator that *Mapping* has to analyze.

6.5.5 Ranking

After creating slices containing candidate nodes and candidate links the respective candidates are *ranked*, i. e. evaluated. The statistical values of the candidate nodes and candidate links retrieved for the *Graph Assembly* step are reused at this stage. The statistical values are normalized by considering the absolute maximum for a specific QoS target[3]. The average values are evaluated as shown in Equation 6.1:

$$s_i(c_k) = \begin{cases} \frac{avg_i(c_k)}{max_i} & \text{iff } r_i = Max, \\ \\ \left(1 - \frac{avg_i(c_k)}{max_i}\right) & \text{iff } r_i = Min \end{cases} \tag{6.1}$$

The *scoring function* $s_i(c_k)$ calculates the normalized score for a specific candidate node or candidate link c_k w.r.t. a QoS target i. Thereby $avg_i(c_k)$ returns the average value of the QoS target i for candidate c_k. max_i represents the absolute maximum for QoS target i. Finally, r_i provides the rank scheme for QoS target i. The evaluation of the scoring function s_i is performed for each candidate node and candidate link in every slice. The scoring values are displayed in the lower right part of the Ranking shown in Figure 6.5. The common ranking semantics adopted by M-TOP maximizes values resulting from the scoring function. Thus, to allow minimization rank schemes, the normalized value is inverted by subtracting it from 1. In the scenario depicted in Figure 6.5, for the QoS target *latency* the normalized scoring values are inverted to get the same ordering as the QoS target *bandwidth* and make them comparable.

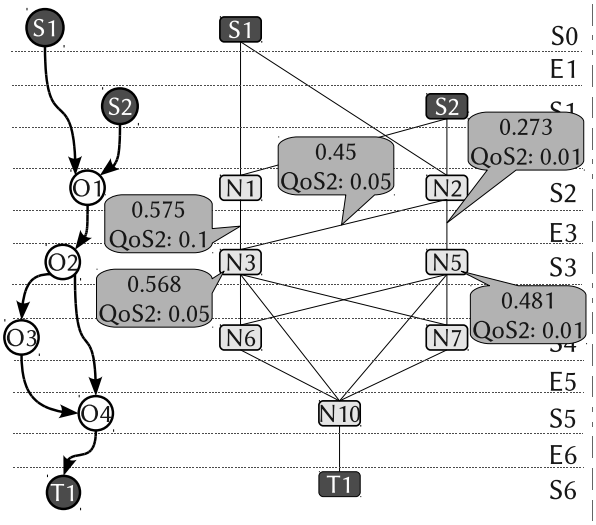
Once all scoring values are determined, the target score w.r.t. the overall QoS target must be determined. The target function is displayed in Equation 6.2:

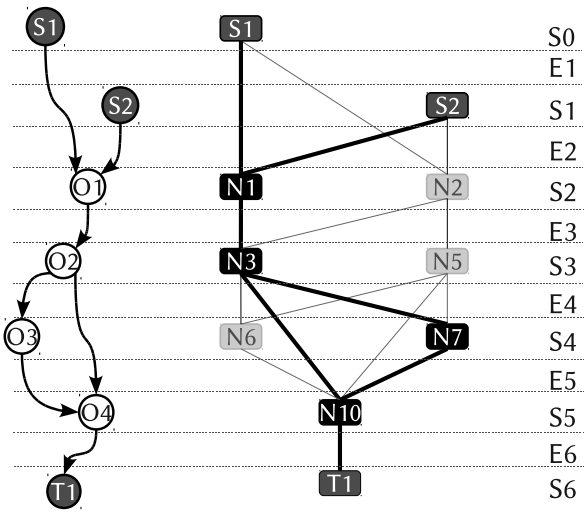
$$t(c_k) = \sum_{i=1}^{n} s_i(c_k) \cdot w_i \tag{6.2}$$

The *target function* $t(c_k)$ realizes a weighted sum by summing up all scoring values s_i for candidate c_k determined beforehand (see Equation 6.1), multiplied by the relative importance factors w_i for the respective QoS target i. The result is a single *target value* for each candidate node and candidate link in every slice. The target values represent the overall performance of a specific candidate relative to the QoS targets defined.

However, two different QoS target classes must be considered: *additive* QoS targets and *absolute* QoS targets. The latter class must match each candidate, i. e. must match all candidate

[3]The maximum for a specific QoS target can be retrieved by a lookup operation from the runtime statistics.





Ranking – Result

- result of Ranking is a **candidate network** of candidate nodes and candidate links
- each candidate has attached the scoring value and a vector representing the **additive QoSs** (in this case consisting only of QoS2 being latency)
- is input for subsequent Mapping

Mapping

- Mapping is **NP-complete** and **complex** → approximate solution
- model as **Constraint Satisfaction Problem (CSP)** and apply **meta heuristic** –genetic algorithm– to solve it
- objective function defines **solution goodness** – the bigger the value the better the solution
- solutions found by M-TOP reaches about **95%** of **optimal solution**

Figure 6.6: M-TOP approach — Ranking-Result and Mapping.

nodes and candidate links. An example for this class is bandwidth, there must be enough for each candidate to satisfy the requirement. *Early Prune* has already discarded candidate nodes that do not fulfill these requirements. The former class must match the whole candidate solution, i. e. must match all candidate nodes and candidate links. An example for this class is latency, since latency depends on all candidates along the critical path of a candidate solution. Thus, the candidate solution in our case must also satisfy additive QoS targets to adhere to the concrete QoS specifications. This is checked by Mapping. It allows to prune candidate solutions that do not match the QoS specifications.

The result is a candidate graph Q_G with target values $t(c_k)$ and additive QoS target values $t_{add}(c_k)$ attached to each candidate link and candidate node as shown in Figure 6.6.

Ranking brings to front candidates, depending on their scoring value, and establishes a natural ordering of the candidates w.r.t. the QoS targets specified. In this sense, ranked candidates are directly comparable to each other.

6.5.6 Mapping

Mapping uses the candidate graph to compute the placement solution from all source operators (in our example **S1** and **S2**) to the sink operator (here **T1**). A potential mapping for the operators is marked by the black candidate nodes displayed in Figure 6.6 on the right side. The utility function displayed in Equation 6.3 must be satisfied.

$$o(C_G) = \max\left\{\left[\sum_{i=1}^{n} t(c_i)\right] \cdot o_{add}(C_G)\right\} \quad \textbf{with} \quad o_{add}(C_G) \in \{0.5, 1\} \tag{6.3a}$$

$$o_{add}(C_G) = \begin{cases} 0.5 & \begin{cases} \text{iff } \forall\ k : r_k = Min \textbf{ and } \sum_{i=1}^{n} t_{add_k}(c_i) > bc_k & \textbf{or} \\ \text{iff } \forall\ k : r_k = Max \textbf{ and } \sum_{i=1}^{n} t_{add_k}(c_i) < bc_k \end{cases} \\ 1 & \begin{cases} \text{iff } \forall\ k : r_k = Min \textbf{ and } \sum_{i=1}^{n} t_{add_k}(c_i) \leq bc_k & \textbf{or} \\ \text{iff } \forall\ k : r_k = Max \textbf{ and } \sum_{i=1}^{n} t_{add_k}(c_i) \geq bc_k \end{cases} \end{cases} \tag{6.3b}$$

The *objective function* $o(C_G)$ for a candidate network C_G, displayed in Equation 6.3a, represents the placement solution with the maximal sum of target values $t(c_i)$ for all candidates c_i (candidate nodes and candidate links) connecting all source operators to the sink operator w.r.t the candidate network. Additionally, $o_{add}(C_G)$ must be fulfilled, i. e. returning 1 iff each additive QoS target k is within boundaries defined by the respective bottleneck condition bc_k (see Equation 6.3b). Without loss of generality, if $o_{add}(C_G)$ returns 0.5, additive constraints are not satisfied and penalize the resulting objective value. This penalty value is a parameter to adjust the actual objective value. This penalty value might differ depending on the problem to solve.

6.6 M-TOP Mapping as Constraint Satisfaction Problem

Mapping is complex. The reasons for this are: Multiple sources must be reached from one or more sinks and cycles in the SP graph (see operators **O2**, **O3**, and **O4** in Figure 6.6) must be solved. Both problems are NP-complete. The former is known as the Steiner-Tree Problem (STP) and the latter is known as the Traveling Salesman Problem (TSP). Solving NP-problems by backtracking approaches is usually computationally too complex. For this reason we decided to model the complex Mapping problem as an instance of a Constraint Satisfaction Problem (CSP) and approximate a solution utilizing a meta-heuristic approach, a genetic algorithm. CSP is adequate to explain the Mapping problem and allows us to apply NP-formalisms to solve it. In the rest of this section, we present the *Mapping Problem* modeled as a CSP instance.

The steps *Early Prune* to *Ranking* are exploited to find a suitable solution for the M-TOP Mapping step, representing a suitable heuristic to this problem.

Next, we present how the mapping step can be modeled as a CSP instance. Then, in Section 6.7, we propose a solution to this problem by using a genetic algorithm approach to solve the Mapping step.

A CSP is described as follows: Given a set of variables V_i with respective domains D_i of possible values ($1 \leq i \leq n$), a set of constraints C_j ($1 \leq j \leq n$), and an objective function O. Each constraint C_j involves a subset of variables V_i ($i \leq n$) and specifies the allowable assignment for that subset. A solution to this problem is an assignment of all variables with a value from their respective domain which does not violate any defined constraint.

The interpretation of the M-TOP Mapping Problem—in the following simply called the *Mapping Problem*—as a CSP is defined as follows:

Mapping Problem: Given an SP graph $Q_G = (O, E)$ with a set of operators O and a set of edges E connecting them, each operator $o_i \in O$ ($1 \leq i \leq n$) is associated with a set of candidate nodes C_i^{node}. The candidate nodes are connected by a set of candidate links C_i^{link}. The candidate nodes and candidate links define the candidate network.

A solution S_i for the Mapping Problem is a mapping that assigns to each operator one candidate node that is going to execute the operator. Thereby, the mapping must satisfy the following constraints:

$$\textbf{(a)} \quad \forall k, l \in \{1, \ldots, m\}: \quad s_{OG}(k,l) = s_{CG}(k,l) = true \ \textbf{and} \ s_{conn}(s) = 1.0 \tag{6.4a}$$

$$\textbf{(b)} \quad \forall k, l \in \{1, \ldots, m\}: \quad S_i(k) \bigcap S_i(l) = \emptyset \tag{6.4b}$$

$$\textbf{(c)} \quad S_i \ \text{is a valid solution } \textbf{iff} \text{ Equation 6.3 is satisfied} \tag{6.4c}$$

Thus, the mapped solution graph (**a**) must form a connected graph according to interconnections defined at SP graph level, i. e. the set of edges E, (**b**) one candidate node must only be assigned once per mapping, and (**c**) a specific objective function $o(C_G)$ must be maximized and the QoS targets must be fulfilled.

According to equations 6.4a to 6.4c, S_i is the i-th solution vector with variables holding candidate nodes and $S_i(j)$ returns the candidate node for operator j with $1 \leq j \leq m$ and m being the number of operators. $s_{OG}(k,l)$ is a function that returns $true$ iff an interconnection between operator k and operator l exists in the query graph, otherwise $false$. Analogously $s_{CG}(k,l)$ returns $true$ iff a candidate link between candidate node $S_i(k)$ and candidate node $S_i(l)$ exists in the candidate graph, otherwise $false$. $s_{conn}(s)$ returns 1.0 iff the solution graph is connected, otherwise a value $1.0 \geq s_{conn}(s) \leq 0.0$ is returned. Note that the return value of $s_{conn}(s) \neq 1.0$ depends on the heuristic applied. Typically a value of 0.0 symbolizes that the solution found is not suitable. In our case, as described in Section 6.7.2, we use a value greater than 0.0 since we just want to degrade the solution's quality but still want it to be considered for subsequent approximation steps.

Figure 6.6 illustrates the *Mapping Problem*. Operators in the SP graph correspond to the variables of a CSP problem. The candidate nodes correspond to the respective variable domains which are grouped in the S-labeled slices. For constraint (**a**), for each interconnection in the query graph there must exist one candidate link in each E-labeled slice, connecting candidate nodes from two adjacent S-labeled slices forming a connected solution graph from the sources to the sink. It is important to recall: S-labeled slices are defined to be adjacent if there is an interconnection between the associated operators at SP graph level. For constraint (**b**), once a candidate node has been selected for a specific operator, the candidate node is not available for subsequent candidate node selections. *Conflation* already merges operators that are executed on the same candidate node. For constraint (**c**), the objective function Equation 6.3a must be maximized.

6.7 Solving the M-TOP Mapping Problem

The *Mapping Problem* is a NP-complete problem and complex to solve. Therefore an approximative solution to the problem seems reasonable by using a search algorithm that heuristically processes the search space. Therefore, a meta heuristic approach, more precisely a genetic algorithm approach, has been selected, since it is widely used and its effectiveness has been proven. Generally, genetic algorithms can be used whenever problems have to be approximated due to the lack of efficient algorithms for exact computation. Genetic algorithms are inspired by the Darwinian law. The fittest individuals, proven to have the ability to adapt best to changing environmental conditions, have a high probability to survive and inherit its genes to subsequent generations of individuals. The Mapping Problem is solved as shown in Listing 6.1.

```
1  // main routine returning the fittest solution
2  def MPGA(num_iterations, k):
3    // initialize population with 20 individuals
4    complete_individuals[] = init_polulation(20)
5
6    while (i != num_iterations && QoS_solution <= QoS_max && best_since <= k) do
7        // assign fitness values to individuals
8        calculate_fitness(complete_individuals[])
9        // select individuals for recombination and mutation
10       selected_individuals[] = selection(complete_individuals[])
11       // recombine individuals
12       recombined_individuals[] = recombine(selected_individuals[])
13       // mutate individuals
14       mutated_individuals[] = mutation(recombined_individuals[])
15       // create individuals for the next iteration step
16       complete_individuals[] = evaluation(complete_individuals[] + mutated_individuals[])
17       // increment loop counter
18       i++
```

```
19
20      // return the fittest solution
21      return solution = select_fittest(complete_individuals[])
```

Listing 6.1: Genetic Algorithm to solve the M-TOP Mapping Problem.

The algorithm runs until either the maximal number of iterations (**num_iterations**) is reached or the best individual is the same (**best_since**) since **k** iteration steps. For both cases the solution's additive QoS targets must be satisfied (**QoS_solution** <= **QoS_max**).

The challenge now is how to implement the single steps of the genetic algorithm to solve the *Mapping Problem* finally. The solution is presented and discussed in the next sections.

6.7.1 Encoding of a Solution and Initial Population

The first step is to define an encoding for the potential solution[4]. It defines the data structure on which recombination and mutation operations work. Individuals are value-encoded having the following appearance: $M_k = \{V_1, ..., V_m\}$. M_k represents an individual of the problem, where k is the index identifying a certain individual within the set of all individuals. V_i represents a value for variable i where each variable is related to the respective operator by the index i which corresponds to the S-labeled slice index with $1 \leq i \leq |S\text{-}slices|$. The domains of values for each variable corresponds to the domain that is defined by the list of candidate nodes for each operator, which is actually also defined by the S-labeled slice with index i. The values are represented by values $c_k^{node} \in C_k^{node}$ representing candidate node IDs, identifying a candidate node. A value is unique within a single solution M_k which means that a candidate node is selected only once per individual (see Equation 6.4b).

The population is initialized randomly (by means of a uniform distribution) to spread the concrete setting of a specific solution, hopefully in the best way. As a starting population, **20** potential solutions are generated. Also another value is possible. However, during our evaluation (see Section 6.8) this value has proven to provide good results with moderate computation efforts. The potential solutions generated in this step are the basis for all subsequent operations. At this point it is not guaranteed that the potential solution generated so far do not violate any constraints as described in Section 6.6.

6.7.2 Fitness Definition and Selection Strategy

The dimension to measure the quality of a certain individual is given by the individuals' fitness values. The fitness value characterizes individuals and makes them comparable to each other. The fitness function calculates the fitness values. The objective function (which is referred to as *fitness function* for the rest of this work) mostly corresponds to the objective

[4]In the following *individual* will be used as a synonym for a *potential solution*.

function defined in Equation 6.3a and satisfies Equation 6.4c. However, Equation 6.4a and Equation 6.4b must be satisfied. Thus the modified fitness function looks as follows:

$$o(C_G) = \max \left\{ \left[\sum_{i=1}^{n} t(c_i) \right] \cdot o_{add}(C_G) \cdot s_{conn}(C_G) \cdot s_{ass}(C_G) \right\} \tag{6.5}$$

Obviously, the solution must also build a connected solution graph. The candidate nodes and candidate links that are part of the solution must build a connected solution graph, denoted by $s_{conn}(C_G)$. $s_{conn}(C_G)$ returns 1 iff the solution graph is connected, otherwise a value 0.5 is returned which degrades the fitness value of this solution. Additionally, $s_{ass}(C_G)$ checks for the unique assignment of a candidate from the domain within a certain solution and returns 1 if the assignment is unique and 0.5, if not degrading the fitness value of this solution. Once we know the fitness of each individual, we have to select some for recombination and mutation where many different strategies exist.

The selection strategy *elitism* saves the best individuals from the current iteration for future iterations. For the remaining individuals a fitness proportional selection strategy seems promising. Individuals having a low fitness value will probably not satisfy additive QoS targets.

6.7.3 Recombination and Mutation

Recombination creates new alternatives in the search space from the individuals selected in the steps performed beforehand. Recombination combines two individuals to form new ones with characteristics from both. The point in question is where to define the recombination points. Here we exploit knowledge from the SP graph level. Recombination points should broaden the search in search space. For this reason, recombination points are defined to be operators representing a branch or a junction. It is expected to have the most impact in terms of search space traversal. The detection of these points is done once per SP graph. Each time a branch or a junction is detected it is marked as a potential recombination point. These potential recombination points are now considered as recombination points to combine two individuals. The decision of which potential recombination point to take is inverse fitness-proportional. Potential recombination points having a worse fitness value are more probable being selected for recombination (and hopefully improve this). Once a recombination point is chosen the individuals are combined and the next two individuals are taken for recombination, until all individuals have been processed. Possibly neither branches nor junctions are detected. In such cases an arbitrary recombination point is chosen by using an inverse fitness-proportional strategy (as with *selection*).

The probability of individuals being selected as *mutation* candidates is proportional to the inverse fitness value. Mutation then occurs with a certain probability (mutation probability) on one variable. The current variable assignment is replaced by a random value of its respective

domain. The mutation rate is equal to $\frac{1}{solution.length}$, thus the mutation rate depends on the population size.

There might be conflicting assignments violating Equation 6.4b by recombination and mutation operations. We assume, enough candidates exist to make more valid assignments than assignments that violate this constraint.

6.8 Evaluation

In this section we present the results of our experimental evaluation. First the system model and foundations are presented. Thereafter the results are discussed.

6.8.1 System Model and Foundations

We implemented and evaluated our approach in a simulation environment that creates virtual candidate nodes and virtual candidate links as well as their respective scoring values. The generated scoring values represent the normalized scoring values as described in Section 6.5.5 and have values between 0.0 and 1.0. For the scoring values we assumed a normal distribution with a mean value of 0.7 and a standard deviation of 0.2 to spread the range of potential values. By using a normal distribution we further assumed that the candidate nodes in our environment are more or less comparable with each other in terms of computing power, although they have different capabilities due to a heterogeneous computing environment. In short, we assumed that about 70% of computing nodes have a scoring value for a certain QoS target that lies between 0.5 and 0.9. The remaining scoring values are located either above or below those values. The respective absolute QoS statistics which are needed to check for violation of additive QoS constraints are calculated to exploit the respective scoring values.

For our tests, we used a notebook equipped with an Intel Core2 T7200 CPU running at 2 GHz and 2GB of DDR-RAM. The population size was 20.

6.8.2 Results

We have evaluated two scenarios, a *simple* and a *complex* one. In both cases we took a real world example of a distributed visualization pipeline. Details can be found in Section 7.5.1. The *simple* scenario consists of a visualization pipeline with a total number of 12 operators, with 7 pinned ones (being either a source or sink operator). The *complex* scenario consists of a total number of 27 operators where only 4 were pinned (also being either a source or sink operator). For all tests the QoS specifications we used are as shown in Table 6.1.

Thus we want a minimum bandwidth utilization of **500 kBit/s** while maximizing this QoS target. A maximum latency (being the additive QoS for our evaluation) of **300 ms** for the simple and **1000 ms** for the complex scenario are required, while minimizing this QoS target.

QoS-target	bottleneck condition	relative weight	rank scheme
Bandwidth	500 kBit/s	0.5	MAX
Latency	300 ms **and** 1000 ms	0.2	MIN
Reliability	50%	0.3	MAX

Table 6.1: QoS specifications used for the experimental results.

Finally, a minimum reliability of **50%** is needed while maximizing this QoS target. The trade-off specifications (relative weights) are **0.5**, **0.2**, and **0.3** respectively. We have created different sets of candidate nodes and assigned a random number of candidate nodes to each operator in each scenario. Operators in one SP graph having the same operator type (e. g., *Render* or *Select*) are assigned the same set of candidate nodes. Not all candidate nodes are suitable to execute arbitrary operators but candidate nodes might appear in multiple candidate node lists. The configurations are generated independently from each other, thus representing different candidate networks. Table 6.2 displays the results of our experimental evaluation.

The *Simple CSP Solver* calculates all possible solutions and its result represents the optimal solution for a given candidate network to compare the results from the M-TOP Mapping with. Thereby, the same candidate network has been used for both approaches. The same objective function, i. e. fitness function (see Section 6.7.2) has been used for both. Configurations only up to 50 candidate nodes have been considered for the comparison with the Simple CSP Solver approach, since configurations with more candidate nodes would have taken an unreasonable amount of time to calculate. Even for this configuration, our test system needed almost 2 days to get this result. Also note the increase in computation time when going from the configuration with 20 candidate nodes to the one with 25 candidate nodes.

	candidate nodes	**20**	**25**	**50**	**500**	**5000**	**50000**
Simple CSP Solver:	*objective value*	0.8225	0.8082	0.8628			
(simple)	*latency (ms)*	245.47	193.32	224.26	*more than **2 days***		
	runtime (ms)	53140	485922	93475625			
M-TOP Mapping:	*objective value*	0.7977	0.7718	0.8282	0.8121	0.8216	0.8566
(simple)	*latency (ms)*	264.13	251.00	211.87	257.69	189.43	268.91
	runtime (ms)	2563	2344	5281	4766	2484	7187
M-TOP Mapping:	*objective value*				0.8523	0.8577	0.8373
(complex)	*latency (ms)*				871.08	922.74	985.39
	runtime (ms)				60000	109265	65797

Table 6.2: Comparison of the M-TOP Mapping approach and a simple CSP solver.

For the simple scenario configurations allowing to compare the M-TOP Mapping approach with the optimum are of interest. Regarding the calculated values as seen in Table 6.2, the M-TOP Mapping (simple) reaches approximately **95%** of the optimal solution determined by the Simple CSP Solver method. For bigger configurations—i. e. M-TOP Mapping (complex)—we achieve similar values for the objective values with adapted additive QoS target values, i. e. 1000 ms for the complex scenario compared to 300 ms for the simple one (see the defined bottleneck conditions in Table 6.1). Thereby, the *runtime* needed by the M-TOP Solver to compute the solutions is faster by orders of magnitude compared to the Simple Solver that computes the exact solution but takes much more time.

For the complex scenario at least 50 candidate nodes are necessary to make a suitable placement. However, M-TOP is designed for much greater sets of candidate nodes. As can be seen in Table 6.2 we get high objective values between **0.84** and **0.86** with an average[5] additive latency value of approximately **925 ms**. Considering the results from the comparison of the M-TOP Mapping with the optimal solution, we assume that those values in average are not worse than the ones collected with the simple scenario. The time needed to get solutions for the complex scenario is higher compared to the ones for the simple scenario due to the higher number of operators to place. It must be pointed out that we have more than doubled the operators to place in the complex scenario.

6.9 M-TOP Supporting Many-to-Many Mappings

Many physical operators and a list of compatible NexusDS nodes might exist for each physical operator that match the respective requirements. This means, there may be many implementation variants for the same logical operator in a SP graph and also many NexusDS nodes the implementations can be mapped to. Our operator placement algorithm called M-TOP is designed for handling an SP graph with one box implementation (rather than many) and many NexusDS nodes the box may be mapped to. Thus, it corresponds to a one-to-many mapping step. To perform a many-to-many mapping step with M-TOP two different possibilities exist. The first one is to perform the one-to-many mapping step multiple times. Thereby, each time a different operator is picked. The solutions are enumerated and the best one is chosen. This approach can be further refined by only performing the one-to-many mapping step for physical operators which seem promising. This might be determined by some heuristic rule set limiting the physical operators considered for mapping. Another way is to slightly modify the M-TOP approach.

Taking into account many physical operators, the encoding of a solution must be adapted. Each solution has the following appearance: $M_k = \{V_1, ..., V_m\}$. This definition corresponds to the solution definition from Section 6.7.1. The difference between both is, that each variable V_i references to a vector of operators instead of a single operator. Also, the value of V_i is a vector of values $c_k^{node} \in C_k^{node}$, where each c_k^{node} represents a candidate node ID. In this context, the

[5] In average according to the measured scenarios.

generation routine for the initial population must be changed as it creates each solution M_k. In this sense, a solution M_k is subdivided into sub-solutions M_k^z, with z being an index for the enumeration of all possible combinations within M_k.

Also, each solution no longer has only one associated fitness value but instead a vector of fitness values: one fitness value for each potential sub-solution M_k^z. The calculation of the fitness values changes in the sense that it must be calculated for each M_k^z. Also, the functions $o_{add}(C_G)$, $s_{conn}(C_G)$, and $s_{ass}(C_G)$ (introduced in Section 6.7.2) must be modified to reflect the fact that the solution encoding look different and thus their evaluation is different.

Finally, the recombination and mutation operations must be adapted to reflect the modified solution encoding. The operations themselves remain the same, but they are applied to M_k^z, i. e. to each sub-solution of M_k.

6.10 Summary

In this chapter M-TOP, a multi-target operator placement strategy of SP graphs for utterly heterogeneous DSPSs was presented. Often, different stream-based applications require dedicated placement of SP graphs according to different QoS constraints. M-TOP finds a distribution for a given SP graph that matches constraints, which in turn are provided by stream-based applications. The operator placement is a crucial operation in DSPSs as it deeply influences the SP graph execution. Unfortunately, constraints might conflict with each other, thus operator placement is subject to delicate trade-offs. Therefore M-TOP defines a heuristic to rate possible solutions. Because the operator placement is NP-complete, M-TOP adopts a meta-heuristic approach in form of a genetic algorithm to solve the placement problem. Thereby M-TOP samples the search space and picks a solution that best matches the application-specific constraints. Our experimental evaluation shows its effectiveness.

Part IV

Tool Support and Conclusion

7 Tool Support and Applications

In the previous chapters a flexible and extensible DSPS has been presented. Such systems often require the use of many different tools in order to perform certain tasks. To create a specific operator for NexusDS several steps are necessary which might require various tools to succeed finally. First, the actual operator is implemented and the respective meta data is modeled. These components are then combined and are made available via the Operator Repository Service in order to work. Then, the developer composes the SP graphs which exploit the (possibly specialized) operators. Once the SP graph is modeled, it is sent to the NexusDS system starting its execution. When execution should end, another message is sent to NexusDS telling it to stop the SP graph execution.

As the example above shows, it is reasonable to provide an integrated tool which supports many—and ideally all—of the above mentioned steps. Obviously, the tool support depends heavily on the actual field of application. This chapter introduces tool support for the complex environment of DSPSs. First, in Section 7.1 the development and employment of an integrated tool support is discussed for the context management platform introduced in Section 2.5.3. After a discussion of the related work in Section 7.2, we introduce the main features of the *NexusDSEditor* in Section 7.3. The NexusDSEditor is the tool developed to support developers during their NexusDS application development process. This section shows how the NexusDSEditor supports the development of context models as well as domain experts, respectively developers during application development. In Section 7.4 a short presentation of NexusDS for mobile devices is given. Section 7.5 then presents a sample application of a distributed stream-based visualization pipeline to render the air flow in buildings. Section 7.6 illustrates how NexusDS could be realized as a streaming service in a cloud environment. Finally, Section 7.7 concludes this chapter with a short summary.

7.1 Motivation for Context Management Tool Support

In the domain of context-aware computing, applications need information about their user's situation to adapt their presentation, actions, or computation. Examples are location-based services [118] such as tourist guides [4, 39], indoor and outdoor information systems [46, 85] and smart environments [76] which present spatially selected information and services on mobile devices, often adapted to the user's current context. Adaption based on context is also done in smart environments such as smart homes where everyday things enhanced with embedded sensors and computing power interact with the inhabitants. Furthermore, adaption on technical layers can be done based on context information, e. g. to switch dynamically to the best available wireless network within mobile communication services [45]. To ease the development of context-aware applications, the management of context information should not be done within the application, but within so-called context models [67] which can also be shared by different applications to decrease the development costs [63] further. In general, context models may contain geographical data (e. g. maps or information about buildings), dynamic sensor data (e. g. the position of a moving object), infrastructure data (e. g. the extent and bandwidth of wireless networks) and context-referenced digital information (e. g. documents or web sites that are relevant in a certain context). In a broader sense, context models can also contain information on technical context such as NexusDS nodes or operators available.

With the growing maturity of sensors, wireless communication and distributed computing environments, pervasive computing enters new application domains. One promising example are production processes in so-called *Smart Factories* [146] where we can achieve huge improvements in transparency and efficiency. Here, pervasive computing not only improves usability or safety, but it really saves money. Different applications may run in a Smart Factory: some may measure the attrition of tools and initiate maintenance processes before critical situations occur [147], while others may track different resources and their state to reduce overstocking of resources [22, 146]. Another example is the monitoring of the overall production process and its energy consumption to improve its overall efficiency, including, e. g. the visualization of air streams in factory buildings, as presented in Section 7.5.1. All these applications have in common that they need context information about machines, workers, and other resources. We illustrate the general problem of context modeling using the application domain of Smart Factories.

The creation of such context models is a tedious task. First, we have to model the *context model schema*, which specifies entities relevant for the application, like machines, tools, etc. Secondly, we have to provide the *context model data*, which represents the concrete instances of specified entities, like the location of the machines, or the state of the tools (e. g. attrition). The context model data can either be static, which means the data is entered once into the system and changed very seldom. Moreover, the data can be dynamic which means it is sensed and provided by sensors in the form of data streams. Finally, the context of interest for the application (higher-level context) must often be derived by rules or probabilistic learning approaches from other context model data (so-called lower level context). To reduce the burden

of obtaining and maintaining such context models, it is beneficial to share the information between applications.

Nexus and NexusDS together form a context management platform. Thereby, Nexus offers access to static context data and NexusDS provides access to streamed context data. The context management platform is an open, federated platform for mobile context-aware applications where arbitrary data providers can make their data available by the platform. A central component is the Augmented World Model (AWM). The AWM is a shared global context model which can be extended by extension schemas. An integrated tool that supports developers of streaming applications is beneficial, when designing new applications or modifying existing ones. It helps to reduce development time and to prevent errors that occur at design time.

7.2 Related Work

An extended survey of current context modeling and management approaches can be found in [24]. One of the most comprehensive approaches for developing context-aware applications is based on the Context Modeling Language (CML) and its associated software engineering framework [67]. It extends Object-Role Modeling (ORM) [66]—developed for conceptual modeling of databases—by features needed for context modeling. CML provides a graphical notation designed to support the software engineer in analyzing and specifying the context requirements of a context-aware application formally. It also extends the *Rmap* procedure of [66] to transform a conceptual schema automatically to a relational schema to manage the context information data in a database. While it is generally possible for applications to share parts of the CML context models, the approach is not intended to scale up for a large number of different applications. The model does not offer direct support for schema evolution and name spaces. In addition no integrated tool that allows the exploration of existing context schemas and context information on distributed servers, as NexusDSEditor does, is available.

There are many tools on the market that could be used for the development of context-aware applications. For creating the context data model, an UML designer such as *Borland Together*[1] or *IBM Rational Rose*[2] can be used. For the XML editing, e.g., for creating concrete context data objects and inserting them into the system an XML editor like *XMLSpy*[3] could be used. Testing the functionality of the system and the defined queries could be done by using a Web Services testing tool like *soapUI*[4]. AWQL and AWML can be regarded as domain-specific languages whose development can be supported by tools like *Xtext*[5] or *Visual Studio*[6]. All these tools are generic solutions and hence are not adapted to the development of spatial, context-aware applications. By using the NexusDSEditor, the overhead of learning how to use the different

[1] **http://www.borland.com/us/products/together/**
[2] **http://www.ibm.com/software/awdtools/developer/rose/**
[3] **http://www.altova.com/products/xmlspy/xml_editor.html**
[4] **http://www.soapui.org/**
[5] **http://www.eclipse.org/Xtext/**, formerly part of *openArchitectureware*
[6] **http://msdn.microsoft.com/en-us/library/bb126327.aspx**

tools is avoided. Furthermore, data exchange between different tools is often difficult or not possible at all. In contrast, the NexusDSEditor is an integrated solution supporting all tasks described above. The user has to use only one tool that integrates all context models, data instances, and interfaces to the context data management system.

For more than a decade, Moving Objects Data Bases (MODBs) have been developed for managing mobile items such as cars. Although management of mobile objects plays an important role for the Nexus system, too, the focus of the research in the area of MODBs is quite different from the context information modeling issues we address with the NexusDSEditor. According to [148], the most important issues in MODB research are location modeling, i. e. representing continuously changing values in a database, adequate query languages and indexes for such data and the representation of impreciseness. Publications on data models or query languages focus on extensions of the Entity Relationship Model (ER-model) for spatio-temporal data [139] or algebras for such data [53]. Impreciseness is discussed in e. g. [7, 37, 111, 138], whereas most publications also present query language extensions for impreciseness. A more recent research topic is data reduction for trajectories [28, 137] which however is closely related to impreciseness. In any case, the focus is specifically on location data, and not on context information modeling in general.

7.3 NexusDSEditor - Integrated Tool Support

The NexusDSEditor supports developers during the development processes of streaming applications. The NexusDSEditor is based on the NexusEditor originally proposed by Nicklas and Neumann [106]. It provides a graphical user interface to interact with the context management platform. In addition, it provides export functions to existing tools, e. g. GoogleEarth. A tool such as the NexusDSEditor can facilitate and accelerate the application development since developers only need to learn a reduced set of different technologies to accomplish the development task. Users do not need to know about XML and XML schema to create a *Nexus Object* of the AWM. Furthermore, such tools offer instant validation and at the same time reduce sources of error.

The original NexusEditor by Nicklas and Neumann [106] supported *schema awareness*, *visualizing geo-spatial data*, *testing with ad-hoc queries or queries by example*, and the *integration of context data*. The NexusDSEditor supports all these features and has been extended by additional ones. The new NexusDSEditor also supports *schema modeling*, *development of application and domain specific operators*, *definition of SP graphs*, *scanning for NexusDS nodes*, and the *introspection of intermediate processing results*.

Next, an overview of how the NexusDSEditor is embedded in the context management platform is shown. After that, the particular functions are presented in more detail in the subsections that follow.

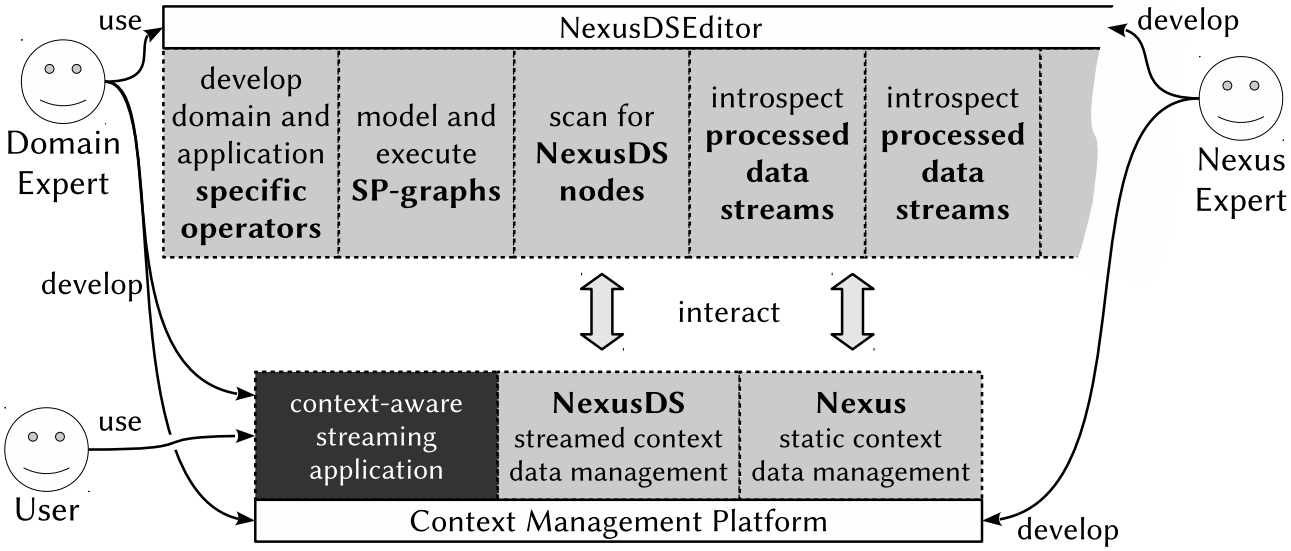


Figure 7.1: Embedding of the NexusDSEditor.

7.3.1 Architecture

Figure 7.1 shows the collaboration of the NexusDSEditor with the *context management platform*. The NexusDSEditor is the central component in supporting the development process and bridging the world of *Nexus experts* and *domain experts*.

As shown on the right of Figure 7.1, Nexus experts develop and maintain the context data management platform as well as the NexusDSEditor. The context management platform consists of the Nexus [103] system and the NexusDS [40] system and was presented in Section 2.5.3. Nexus experts also develop the main extensions for the NexusDSEditor. The NexusDSEditor consists of different extensions such as the extension to *model and execute SP graphs*. Additional extensions can be plugged into the NexusDSEditor.

On the left, domain experts use the NexusDSEditor functionalities to develop context-aware streaming applications and context data management platform extensions respectively. Context management extensions hereby include additional services for NexusDS and extended schemas for the AWM. With the NexusDSEditor, most potential conflicts are recognized and can be eliminated at design time. Such a conflict might arise if connected slots of two operators are not compatible with each other. Thus, compatibility of interconnections between operators needs to be checked at design time to guarantee that SP graph is valid already at design time.

Finally, users use context-aware streaming applications and access functionality and data which domain experts have developed before.

7.3.2 NexusDSEditor Functions

In the following, the main features of the NexusDSEditor are presented in more detail. The features of the NexusDSEditor support developers during their development task. Thereby the

NexusDSEditor beside others supports schema modeling, context data integration, development of application-specific and domain-specific operators, or SP graph modeling and deployment. These features reduce problems due to faulty modeling by checking for correctness at design time. Each subsection corresponds to one of the features introduced in Section 7.3.

7.3.2.1 Schema Awareness

The NexusDSEditor is schema-aware, which means that the NexusDSEditor reads the schema of the underlying context model and therefore knows (and checks) which data objects are valid, which attributes they may have, and whether they are required or optional. When a user creates a new data object, the NexusDSEditor offers a list of available object types or the current schema. The same applies for adding or changing attributes or attribute values within objects. In addition, the correct data type of the attributes (e. g. string, polygon, or number) is checked, and appropriate editing modes are offered. For example, for inserting and changing spatial information, a graphical tool exists which also allows to display a satellite picture as visual reference for the geometries. The NexusDSEditor can read both the AWM Standard Class Schema and additional extended class schemas, and distinguishes between those definitions by using the correct namespaces of the XML schemas.

7.3.2.2 Schema Modeling

Extending the existing schema information using the NexusDSEditor is quite comfortable. As can be seen in Figure 7.2, the schema browser is activated. That function is used to browse for already existing object models (classes) (1). In this example, we browse a class schema containing information about a smart factory environment. The *Tool* class is selected (2). A *Tool* is geo-referenced and has a certain position at a certain point in time. The advantage of geo-referenced tools is that overstocking of tools can be avoided and they can be localized and thereby found easily by the workers [146]. A more detailed view (telling the arbitrary and optional attributes it has) of the selected class is provided in the bottom right corner of the graphical user interface (GUI) (3). By simply using *Create subclass* (4) a new class *ScrewTool* is created inheriting all superclass attributes.

A new tabbed window opens as can be seen in Figure 7.3. Now, the newly created class can be enriched with additional optional and required attributes (5). Here, only non-conflicting attributes can be added, i. e. attributes which are not already assigned to the *ScrewTool* class to be created, or one of its super classes. This increases the productivity since potential conflicts are recognized and can be eliminated at design time. Furthermore, it is possible to add already existing classes to the list of super classes for the *ScrewTool* class (6). This results in multi class inheritance. In this case, the attributes from these classes are also inherited by the current class. Afterwards, the *ScrewTool* class can be assigned to an existing schema or a new one can be created (7).

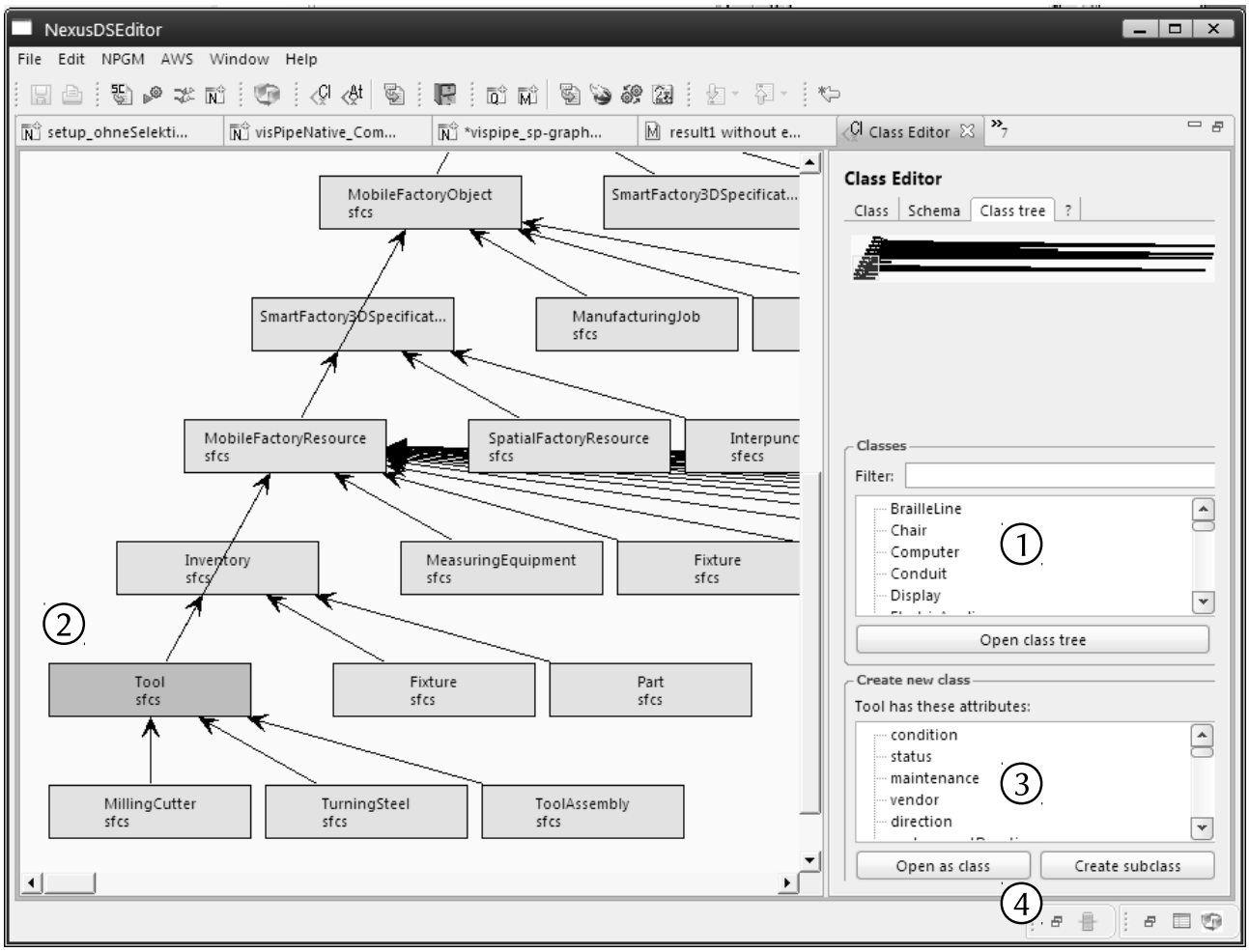


Figure 7.2: The NexusDSEditor class browser.

7.3.2.3 Integration of Context Data

Once the schema information is created, a request to update the schema information in the Context Management Platform is generated. The schema is sent to the Nexus expert who makes the information available within the AWM. Now, the context data can be modeled by the domain expert using the same view as the result set browsing (see Figure 7.4). Once modeled, the context data can directly be sent to the Context Management Platform and inserted on a Context Server (CS). Then the data can immediately be accessed and used by context-aware applications.

7.3.2.4 Visualizing Geo-spatial Data

To visualize geo-spatial data (e. g. the results of an AWQL query), the NexusDSEditor offers two different ways: Internal data visualization and external data visualization.

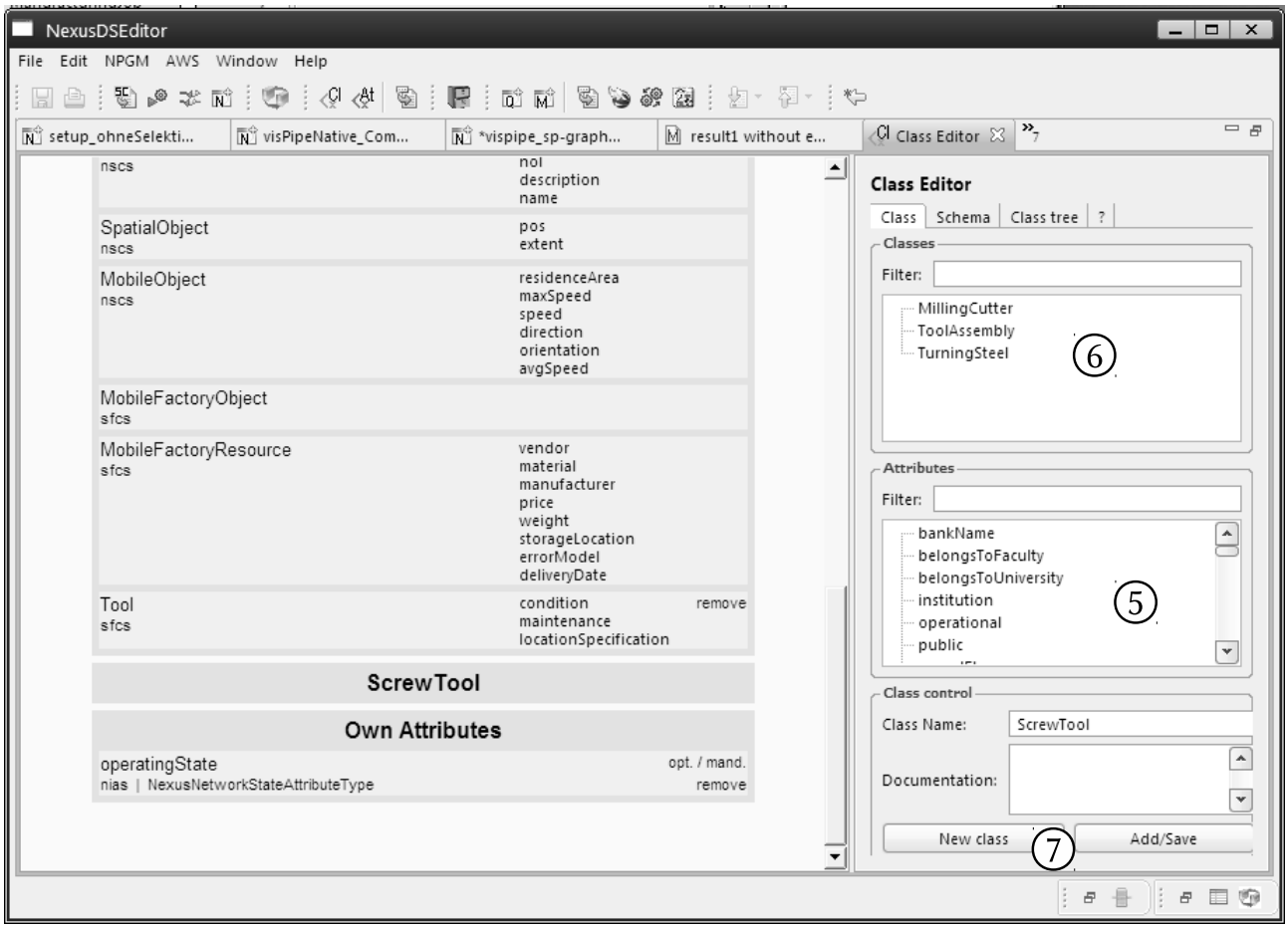


Figure 7.3: The NexusDSEditor class composer.

Internal data visualization Once an AWQL query is created, it can be sent to a given server endpoint, i. e. a CS or a federation node of the Nexus system. The NexusDSEditor displays the response in plain XML format to the user. The response can then be parsed by the NexusDSEditor which displays it in a tree view as shown in Figure 7.4 (1). The user can now browse though this representation. When an object or attribute value is selected by the user, the right side of the window shows appropriate information about the selected part, e. g. a graphical representation based on its type from the corresponding schema (2). Data objects can be added, removed, or modified in this view. Additionally, the whole result set can be viewed in a map representation that draws the extent of all objects of the result set, if available. To visualize these objects in their spatial environment, a background picture can be loaded and mapped to the coordinates of the result set objects. With this, the user can match a given spatial data set with a satellite image or digitalized map and use this to correct errors or insert missing objects.

External data visualization TheNexusDSEditor also offers a bridge to existing tools like GoogleEarth[7]: A predefined data set that comes from the GoogleEarth server can be aug-

[7] **http://earth.google.com**

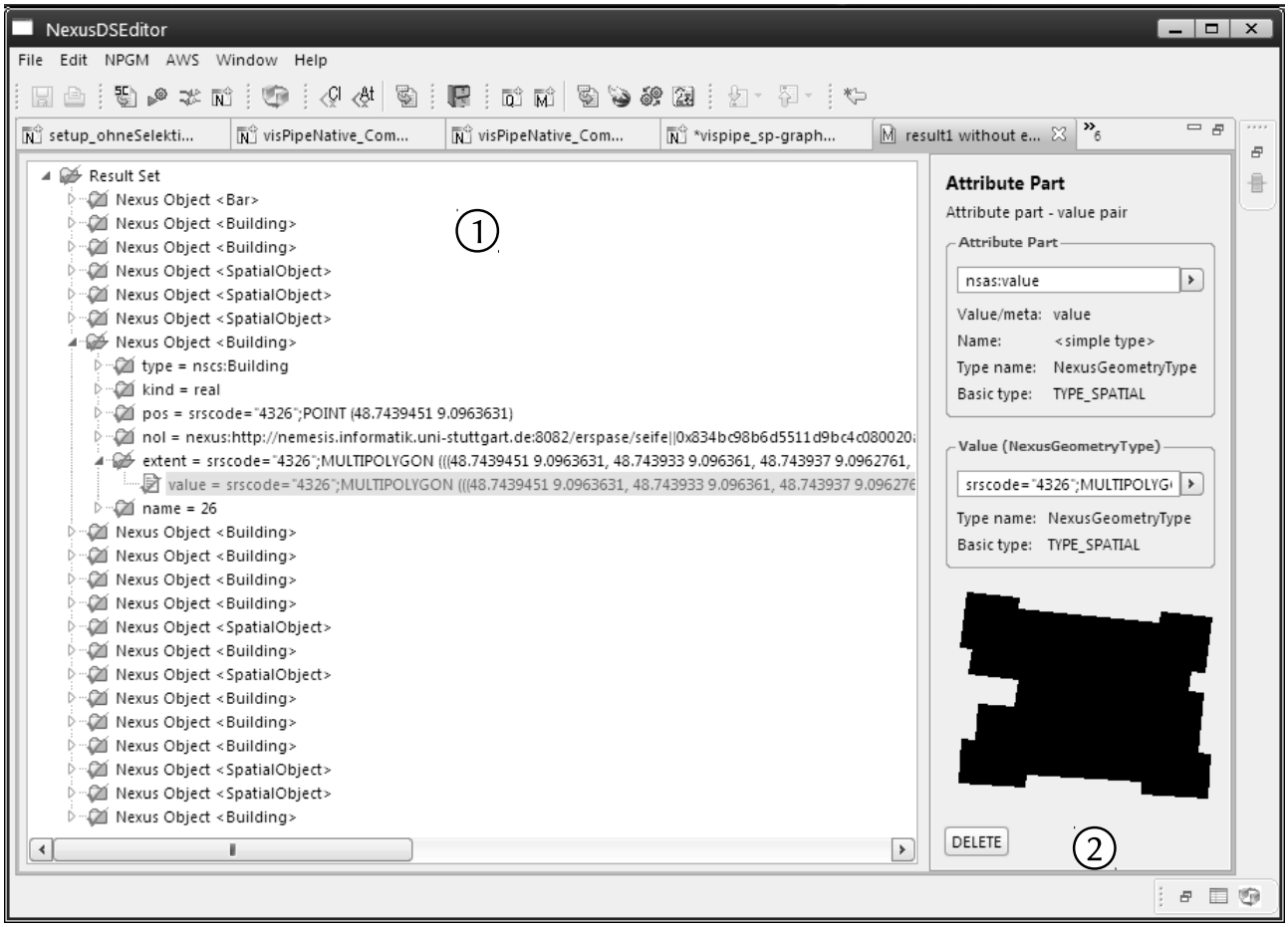


Figure 7.4: Schema-aware result set composer in NexusDSEditor.

mented by user-defined data using the Keyhole Markup Language (KML). KML is an XML derived format that allows exchanging a list of points of interest, so-called placemarks. These placemarks can be structured into folders and documents but normally only have a name, a description and geometric and style information. Since GoogleEarth is used as a visualization tool and not for storing or querying information, the NexusDSEditor puts all attribute values in a human readable format into the description element of the placemark, so that the complete information is shown when the user selects an object in GoogleEarth. As the Context Management Platform and GoogleEarth support the same basic geometric elements (points, lines, polygons, and collections of these elements), the spatial attribute values can be easily exported from AWML to KML.

7.3.2.5 Testing with Ad-hoc Queries or Queries by Example

The NexusDSEditor provides two ways to create an AWQL query: ad-hoc (created directly by the user) and query by example (created from a given data object).

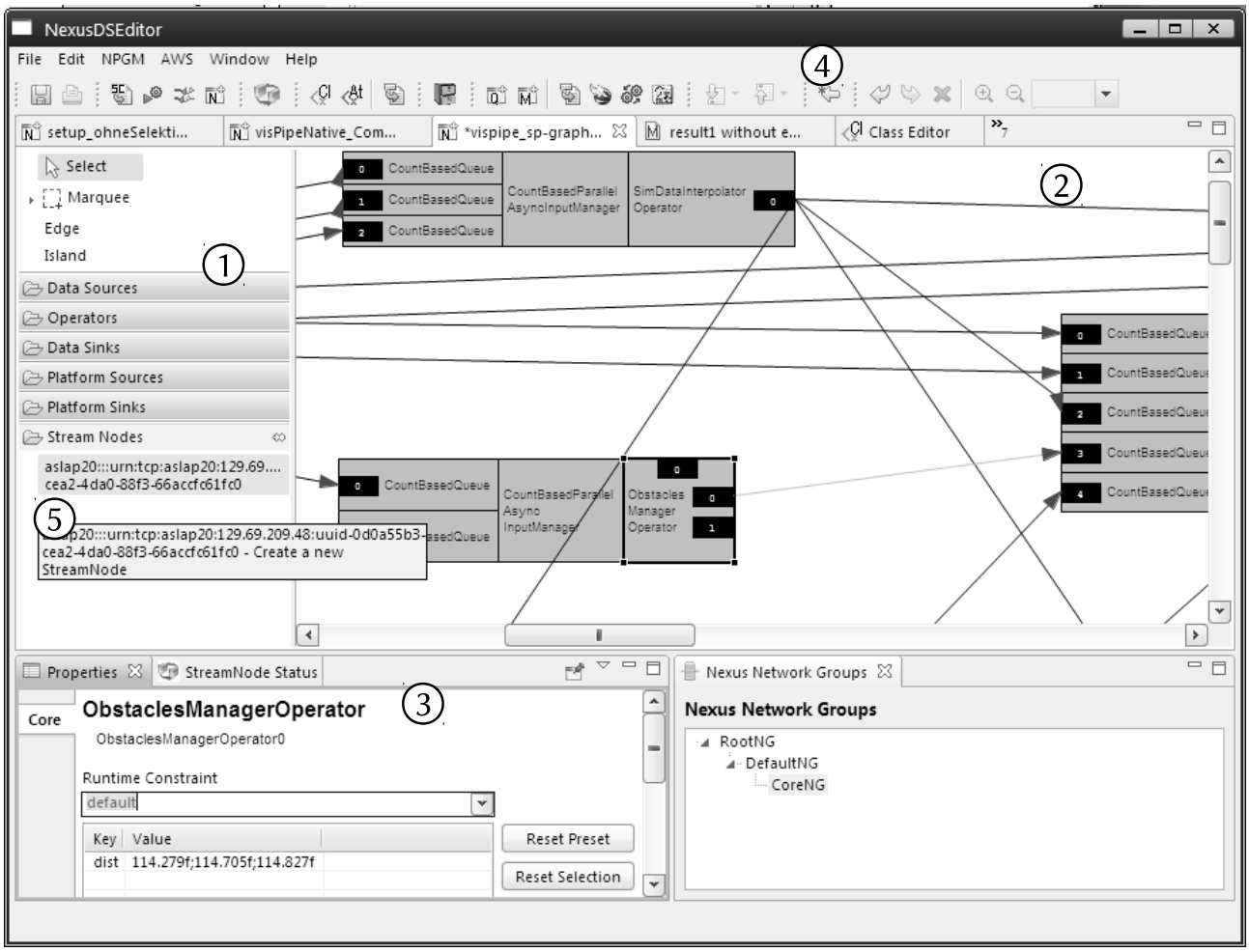


Figure 7.5: SP graph modeling window of the NexusDSEditor.

Ad-hoc queries When the user inserts a restriction operator, the NexusDSEditor checks for the correct reference values: For example, when inserting an overlap operator to restrict the result set to a given area (also known as window query), the reference value has to be a valid geometry, e. g. a polygon. For this, the NexusDSEditor again offers a graphical input tool.

Query by example To create a query by example, the user can select or create a data object as template. The NexusDSEditor can then create a query that contains all attribute values of that template object as restriction parameters. Therefore, the attributes are taken as restriction target and the corresponding values as restriction value. E. g. an object containing the attributes **name** and **opened** with the respective values **EM-Cinema** and **true** will result in a query with the restriction term **name=EM-Cinema AND opened=true**. The generated query can now easily be modified by removing parts of the restrictions not needed. Only objects that are similar to the template object in the remaining aspects are queried.

7.3.2.6 Definition of stream queries

Figure 7.5 shows the SP graph modeling GUI of the NexusDSEditor. At (1) is the toolbox offering different operators. These operators are the basic parts of which an SP graph is composed (2). This is the drawing area where the SP graphs are modeled. Each operator represents a certain operation on the streamed data. On area (3) the properties of the currently selected element on the SP graph modeling area are displayed, in this case the *Obstacle Manager Operator*. The outgoing data channels from the selected element are highlighted to facilitate the modeling process. The NexusDSEditor offers an option to define so-called *isles* which correspond to the conflation principle described in Section 6.5.2. Hereby, the developer can specify which operators are executed on the same NexusDS node (this scenario is displayed in context of the real-world example in Figure 7.6). The final result is an SP graph which is arranged into isles. Finally, some shortcuts are offered to mostly used functionality grouped within a toolbar (4).

7.3.2.7 Development of new domain and application specific operators

Developers have the opportunity to integrate specific operators in NexusDS. Therefore, the developer has to provide the actual implementation of the physical operator and also a set of meta data that describes the operator properties. To satisfy this requirement, the NexusDSEditor provides a modeling component that allows to model the required operator meta data easily and to package the operator for later deployment. In this way it provides a modeling view as shown in Figure 7.4 to model the operator-related meta data. Thereby, the schema-awareness of the NexusDSEditor guarantees that the modeled data is correct. The newly modeled meta data is then packaged together with the actual implementation, which is done with a corresponding development environment such as *Eclipse*[8]. Also, possible third-party dependencies are collected and an operator package is created. After that, the operator package can be inserted into the Operator Repository Service (ORS).

7.3.2.8 Scan network for available stream processing nodes

In NexusDS, stream query graphs are processed in a distributed fashion. Therefore, the SP graph is distributed and executed across different NexusDS nodes. The NexusDSEditor supports scanning the network for available NexusDS nodes and present them to the developer. In a second stage, the developer can pick the NexusDS nodes he prefers and assign them to the corresponding isles. The NexusDSEditor offers the possibility to define the deployment and runtime constraints of a SP graph fully. Thus, the SP graph (in fact a physical NEGM SP graph as described in Section 4.5.1) is fully defined and ready for deployment. The NexusDSEditor therefore offers a deployment functionality that deploys the isles to their associated NexusDS nodes, thus initiating the execution of the SP graph.

[8] **http://www.eclipse.org/**

7.3.2.9 Introspection of data processed by the stream query graph

After the SP graph has been deployed, it is executed and streamed data is processed accordingly. Nevertheless, errors may occur during operator development. Therefore it is beneficial for developers to analyze processed data that is transferred between operators. The NexusDSEditor can be exploited as a testing environment for newly developed operators before utilizing them in a productive environment. For this purpose, the NexusDSEditor supports a special kind of operator class named *Visualizers*. Visualizers consist of two components: One component for connecting to the SP graph to retrieve the corresponding intermediate streamed data resulting from the operator. The second component is a visualization component which is plugged into the NexusDSEditor to actually display the intermediate streamed data within the NexusDSEditor.

7.4 NexusDS for Mobile Devices

In order to integrate mobile devices to work within NexusDS we have also implemented a version for mobile devices. To be more specific, we have implemented a version for mobile devices running Java 2 Micro Edition (J2ME) . Thus, mobile devices (called mobile NexusDS nodes) become a part of the NexusDS system as they are discoverable and addressable. However, the functionality of these mobile devices has some shortcomings. First, arbitrary services cannot be instantiated on the device. This means that service packages cannot be loaded from a remote site and instantiated while the NexusDS system is running on the device. The reason for this is that the J2ME version does not provide this kind of mechanism. Secondly, the operators available are limited to operators already existing on the mobile NexusDS node. This means, additional operator packages cannot be instantiated on these devices as they lack providing a proper loading mechanism. Moreover, there must be appropriate operator implementations for the target platform. Thirdly, the features of J2ME compared to Java 2 Standard Edition (J2SE) is far more restricted, e. g. no custom class loaders are allowed. Furthermore, the J2SE version represents a single package with a well-defined set of features. In contrast to this, the J2ME version is subdivided into two categories supporting only a subset of these features. The smallest category J2ME/Connected Limited Device Configuration (CLDC) represents the set of features all devices running Java applications must support, and is intended for devices with reduced capabilities. The available features, however, can be extended by device-specific profiles and optional packages. The category J2ME/Connected Device Configuration (CDC) is intended for devices running a fully-fledged Java Virtual Machine (JVM) but still with limited features compared to the J2SE version. The most important category for J2ME is the CLDC category, as many devices (mainly mobile phones) support this standard.

The shortcomings mentioned above enforce the NexusDS system to be tailored to these devices and the scope must be known before the NexusDS system is started on these devices. Thus, all services and operators must be installed manually and made available before mobile NexusDS node startup procedure begins. This means, that no additional operators or services

can be instantiated or started that have not been installed beforehand. We have implemented source operators that can extract sensor data originating from the sensors of the mobile device. Furthermore, sink operators capable of displaying incoming data streams of images have been implemented. However, the only service currently running on such devices is the Operator Execution Service executing operators. This means, that source operators can be executed and thus the data present on the mobile device can be integrated into data processing. Sink operators representing the final destination for the processed data are also executed here, providing a minimalistic functionality for data processing. Due to the missing processing power and the energy-related issues with such devices, more performance demanding services such as the Core Graph Service are currently not running on these devices. The planning of the deployment and execution of SP graph is performed on less restrictive NexusDS nodes.

7.5 Sample Applications

In order to evaluate NexusDS, sample applications have been implemented. Thereby, the NexusDSEditor is utilized to create specific operators, model the context data, and to model the SP graphs defining how streamed data must be processed. Section 7.5.1 presents the first application which is a distributed and context-aware visualization of the air flow in buildings through streamlines. Then, a second sample application is presented in Section 7.5.2. This application is a tool for supervising modern production environments such as the ones of the Smart Factory [146].

7.5.1 Context-aware Streamline Visualization

The context-aware streamline visualization is a complex distributed stream processing scenario. It is described in more detail in Section 2.3.1. In short, the scenario calculates and renders the air flow in buildings. This may enable optimization potential for production environments such as the Smart Factory [146]. By knowing the air flow through the production environment it is possible to optimize the cooling of the producing machinery. Thereby it heavily relies on operators belonging to the domain of visualization. However, from a DSPS, and in this case especially from a NexusDS point of view, these operators are highly domain-specific to the domain of visualization. The resource requirements of the visualization process might also be constrained, meaning that depending on the target of interactivity different configuration scenarios might be necessary. Augmented reality applications might need a higher level of interactivity as a web-based monitoring application would, for example. This in turn influences potential operator placement and so forth.

The scenario depicted in Figure 2.2 represents a simplified version of the actual distributed and context-aware visualization pipeline. The real-world scenario which was actually implemented is displayed in Figure 7.6. Obviously, it is quite complex, and crafting this by hand without adequate tool support is cumbersome. In order to implement the scenario, the func-

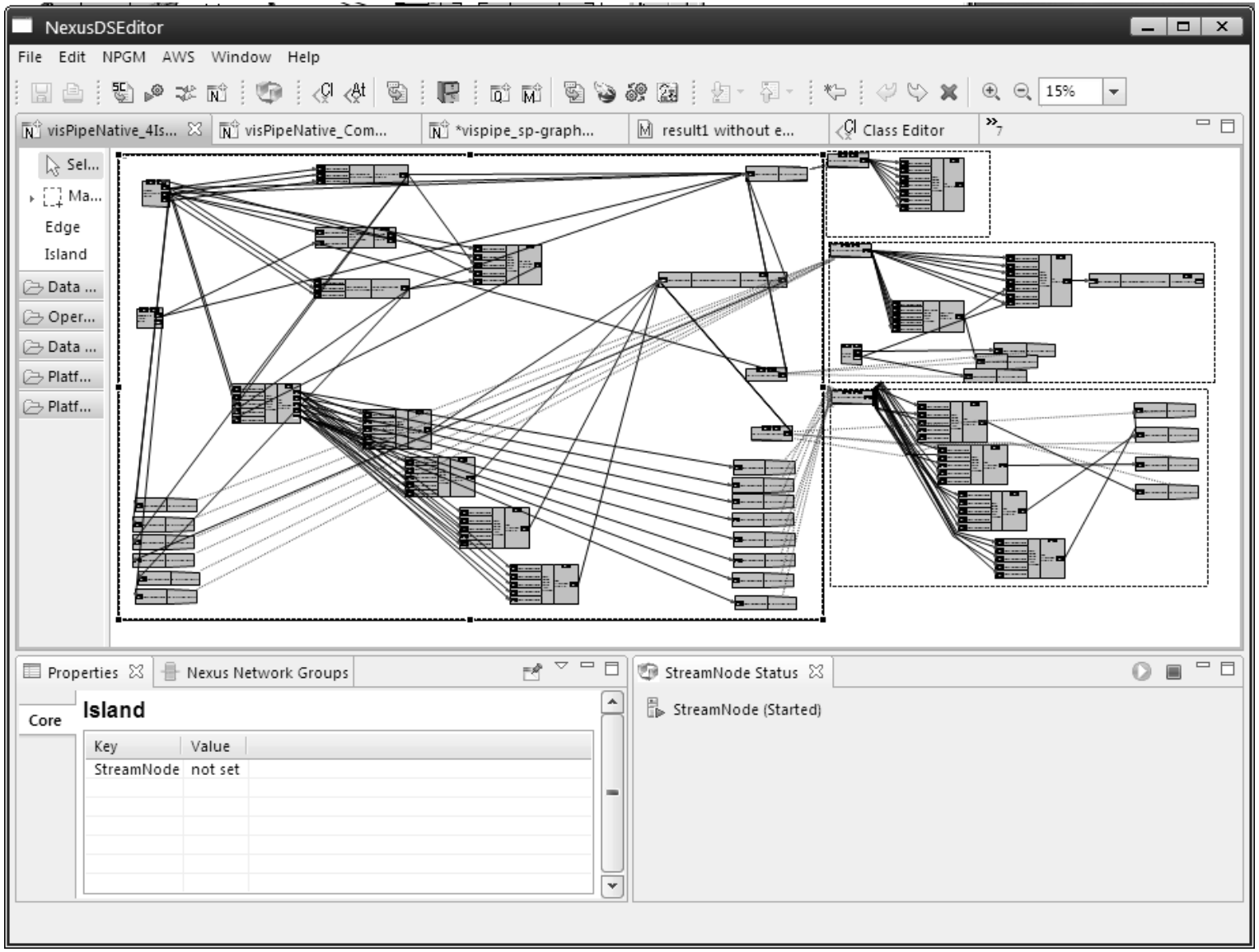


Figure 7.6: Real-world SP-graph for the streamline visualization pipeline scenario.

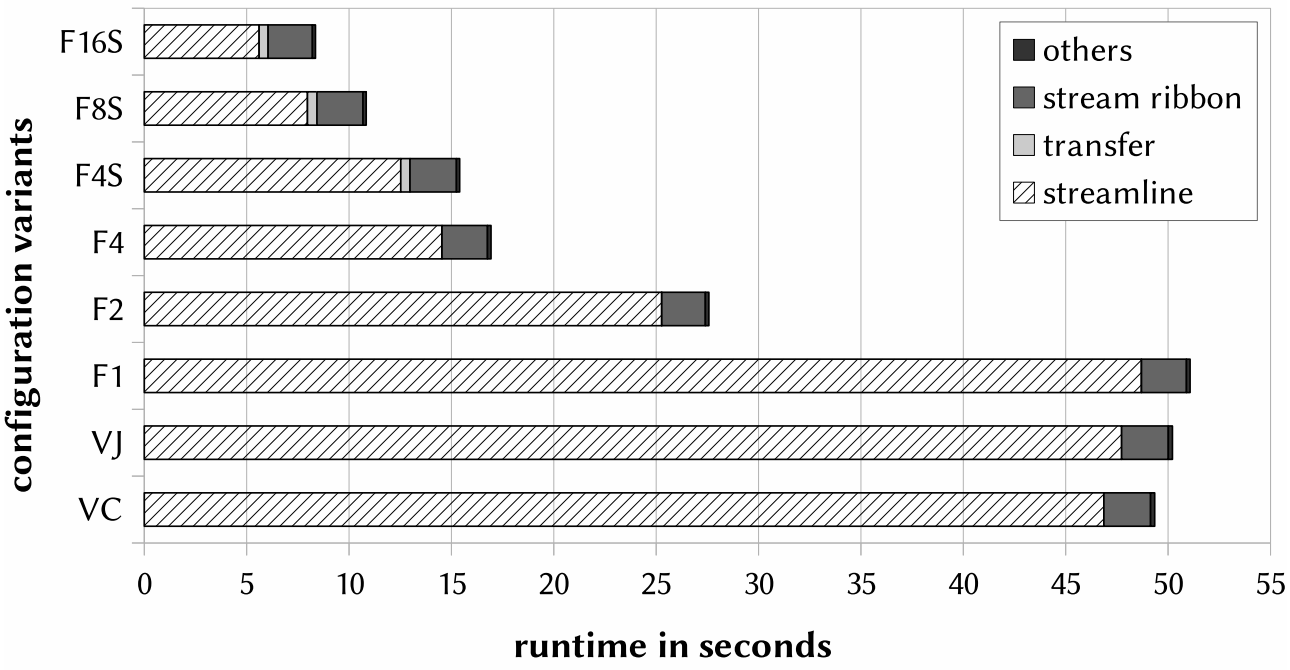


Figure 7.7: Measurements of the context-aware visualization pipeline with different configuration variants and different time fractions.

tionality of the formerly presented NexusDSEditor has been exploited. The specific operators have been developed using a corresponding development environment. In this case, a combination of *eclipse* (Java) and *Visual Studio* (C/C++) is used. In fact, the specific operators have been developed in Visual Studio and implemented in C/C++. Then, they have been integrated into NexusDS which is Java-based. The communication between the C/C++ and the Java world, i. e., the transfer of the data from Java to C/C++ and vice versa needed special care and has been solved as described in [117]. The real-world example comprises 20 operators in total (excluding the platform source and platform sink operators needed for the network transfer of data), about 115 interconnections between the different operators, and has about 3660 lines of XML code in total.

Figure 7.7 shows the runtime fractions of the experiments conducted with different configuration variants. These experiments were conducted on a test platform consisting of PCs with Core 2 Quad Q6600 CPUs. Each PC was equipped with 4 GB of DDR2 RAM. The single PCs were interconnected by a gigabit ethernet connection.

In order to get a representative comparison, a variant called **VC** has been implemented which constitutes the monolithic version of the context-aware visualization pipeline purely written in C/C++. It represents a single executable which performs the visualization tasks. In total, a runtime of approx. **49s** was achieved. A Java-based monolithic implementation of the visualization pipeline was also implemented, denoted by **VJ**. For this configuration a total runtime of approx. **50s** was achieved, which means an increase in runtime of about **2%**. Thus, the transfer times of the C/C++ data structures to and from Java is relatively low. In order to

get an impression of the overhead introduced by NexusDS, a third monolithic configuration variant **F1** was created. In this case, an equivalent SP graph was created where all operators are executed on the same NexusDS node. This configuration reached a total runtime of approx. **51s** and results in an increase in runtime of about **3%** compared to the original variant **VC**. For this configuration variant too, the overhead introduced by the Java-based DSPS NexusDS is relatively low and thus negligible.

As the experiments have shown, streamline calculation is the most consuming operation. Thus, for the configurations **F2** and **F4** the streamline calculation has been parallelized by exploiting the NexusDS capabilities.[9] As the results in Figure 7.7 show, for **F2**, which executes two streamline calculation operators in parallel, a runtime of approx. **28s** was achieved. For **F4**, which executes four streamline calculation operators in parallel, a runtime of approx. **17s** was achieved. Thereby, each operator was executed on a different core exploiting multi-core systems. So, parallelizing streamline calculation enormously speeds up execution. To parallelize and thus further reduce the runtime for a single calculation step, the visualization pipeline needs to be distributed. Therefore, three additional configuration variants have been developed, called **F4S**, **F8S**, and **F16S**. The variant **F4S** is the same as **F4**, with the difference that the rendering and stream ribbon calculation was transferred to a separate NexusDS node. This optimization resulted in a slightly better total runtime of approx. **15s**. Finally, the variants **F8S** and **F16S** execute eight and sixteen streamline calculation operators in parallel respectively. The configuration **F8S** achieved a total runtime of approx. **11s** and **F16S** achieved a total runtime of approx. **8s**.

The experiments show that the overhead introduced by NexusDS is relatively low, although the visualization application is a native application written in C/C++ and not in Java as NexusDS is. An additional advantage is that the single operators can be reused in different scenarios and plugged together differently. This process is supported by the NexusDSEditor presented in Section 7.3.2.

7.5.2 Nexus Explorer

The *Nexus Explorer* is a monitoring application in the context of the Smart Factory. Its main architecture is depicted in Figure 2.3 in Section 2.3.2. The Smart Factory with all its facilities, machines, and workers are modeled as part of the AWM. The Nexus Explorer allows to monitor current activities in the Smart Factory. This is important for facility managers to, e. g. detect potential problems in time or just to get an overall view of the current state of production. Furthermore, it supports workers in locating the position of tools or querying the status of tools currently in use. Beyond this, the main function of the Nexus Explorer is to help workers and experts in solving production failures along the assembly lines swiftly. In this regard, the Nexus Explorer allows to rewind the production data collected during production

[9] Parallelization would also be possible with a monolithic application. However, the scale-out factor is mainly limited by the locally available resources, in our case the available cores. With NexusDS we have no local boundaries.

and to replay it to find the failure's origin. The historical factory data therefore is stored in the context management platform. This enormously helps with failure detection, e. g. when production is faulty and the inherent problem is not obvious.

Figure 7.8 shows a capture of the Nexus Explorer. Its main features are as follows:

(1) Once a failure occurs during production, it is important for the employee who analyzes the faulty situation to be able to travel back in time. Thereby, the slider is moved in the appropriate direction and the corresponding data is shown by different plugins.

(2) This plugin represents a map plugin which visualizes the position of different objects in and around the manufacturing environment. More fine grained information is provided by the attribute plugin (3).

(3) The attribute plugin shows more detailed information on the currently selected object from (1). E. g., it shows the status of a tool or the exact current location of an object.

(4) An important feature is to limit the visible objects to reduce complexity. With this functionality the employee is able to restrict the displayed objects to only those which are considered important. E. g., he can restrict the visible objects only to machines or tools and hide unnecessary details.

The Nexus Explorer has a modular architecture which allows to integrate additional plugins. A plugin developer can add specific plugins unique to their manufacturing environment. With the Nexus Explorer it is possible to monitor ongoing activities in the Smart Factory in real time. The Nexus Explorer also allows to access historical data of the production facilities. The Nexus Explorer allows to rewind the historical data stored in the context management platform and replay the collected data. This enormously helps in failure detection, e. g. when production is faulty and the problem is not obvious.

7.6 NexusDS as a Streaming-Service

In times of cloud computing, a service that allows the stream-oriented processing of data streams seems a useful and promising field of application. Moreover, a stream-based cloud service opens up a broad range of future applications. We could imagine a context-aware cloud-based navigation application which renders a map of the nearby surroundings and reacts to the user's current context. At the same time, it is possible for the service to collect data streams originating from many different sources and process them in real time, thus offering fast responsiveness and being always up to date.

Thereby, the cloud-oriented NexusDS service is modeled as a distributed service oriented system, which allows to process the data in a distributed fashion. The architecture and construction of NexusDS is well suited to fit in a cloud-based environment, as it is distributed and service oriented. Thus, we can imagine NexusDS offering Streaming as a Service (StaaS). To the outside it offers a simple querying interface which allows users to invoke the service by

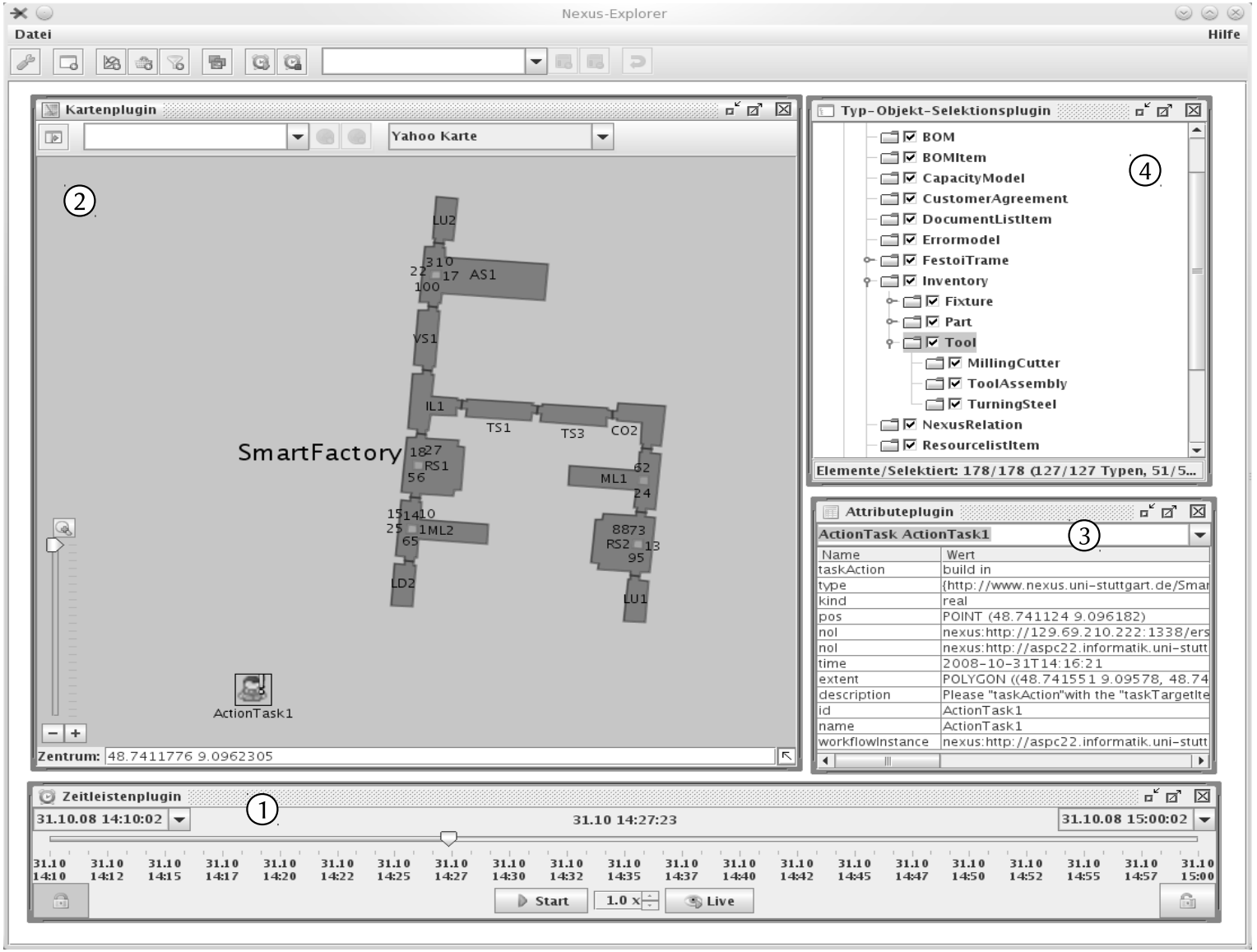


Figure 7.8: Screenshot of the Nexus Explorer.

passing an appropriate document and receiving the results. The streaming service itself consists of a broad variety of services which are hidden from its users. These services in turn are cloud services, too. The service-oriented modularity guarantees a flexible utilization and extensibility of the cloud-based streaming system. Thus, a provider could enhance the cloud-based NexusDS service by additional services that fulfill a predefined functionality.

NexusDS could also be applied to scenarios other than the cloud-based one. An example is the utilization of NexusDS as a monitoring and surveillance system in a *Smart Factory* [146] environment. In this case, contrarily to the scenario from Section 7.5.2, NexusDS is attached to a Manufacturing Service Bus (MSB) [97] and integrated into the existing service landscape. The main motivation in integrating a DSPS into manufacturing environments is to bridge the execution environment of EAI processes running in the manufacturing environment, namely the MSB, and the modeling of the EAI processes. Thereby, the DSPS analyzes the data originating from the manufacturing environment and provide modeling suggestions for future EAI scenarios, building a feedback loop.

The integration of NexusDS in manufacturing environments with a MSB is sketched as follows. The first thing is to attach the NexusDS system to the already existing system which stores the status of each completed assembly operation in a database, possibly also including the assembly errors. In the following, this system is referred to as *supervisor system*. Therefore a source operator is developed which receives updates from the supervisor system. NexusDS then transforms and processes the incoming data streams according to a deployed SP graph. The resulting data is then written to some output which could be a repository to store the manufacturing data or a web interface visualizing the current state of the monitored production environment. The SP graph might be deployed to NexusDS via calling a *Web Service* interface which is attached to the MSB. This permits the integration of NexusDS into the control flow environment represented by the MSB.

7.7 Summary

In this chapter we presented the NexusDSEditor, which provides all the support needed for designing and populating context models for the context management platform. Thereby the NexusDSEditor supports the schema design for newly created context models, supports extending existing schemas for usage in new domains and supports the deployment of the schemas to a context server. After designing the schemas, the NexusDSEditor offers ways to create context data and to store it in the Nexus system. Furthermore, the NexusDSEditor allows to test and visualize AWQL queries needed for the development of context-aware applications. The main advantage of NexusDSEditor is that all functions supported are integrated in a single tool that is tailored to develop context models and NexusDS related components as well as SP graphs for data stream processing. Additionally, NexusDSEditor helps to avoid errors and to speed up the development process. The schema-awareness of the NexusDSEditor helps to avoid syntactic errors in the modeled context data. The highly flexible NexusDS system requires

descriptions of the technical context, which can also be provided using the NexusDSEditor. Compared to more generic, conventional XML editors or modeling tools, the NexusDSEditor supports additional features specific to the context management platform, e. g. geometries can be created or edited and multi-types can be handled. Domain experts are relieved from directly editing XML files. Beyond, the NexusDSEditor provides a linking to GoogleEarth in order to visualize AWM objects in a larger geographical context.

Sample applications that were implemented for the presented context management platform consisting of Nexus and NexusDS were presented. First, an application which visualizes the streamlines in buildings was shown. The particular challenge here was to integrate C/C++ based operators into the Java-based NexusDS system. The application demonstrates that the overhead introduced by NexusDS is negligible and the performance is an acceptable alternative to the monolithic pure C/C++ implementation. However, with NexusDS parallelism and distributed execution of the visualization pipeline can be exploited. This results in an increased overall performance and a reduced time to compute the streamlines of the scenery.

The second application is called *Nexus Explorer*. It is a monitoring application in the context of the Smart Factory. Therefore, the Smart Factory is modeled within the AWM with all its machines, workers and so forth. With the Nexus Explorer it is particularly possible to monitor ongoing activities in the Smart Factory and locate the current position of tools or query its current status. Beyond this, the Nexus Explorer allows to rewind the processed data which is stored in the context management platform and replay the data. This provides handy failure detection, e. g. when production is faulty and the original problem is not obvious.

This chapter is concluded by a short discussion on how NexusDS can be applied to the field of cloud computing and how NexusDS can be integrated into manufacturing environments via an MSB. This however does not represent the status quo of research but is more like a sketch on how NexusDS can also be utilized in environments such as MSBs and cloud computing.

8 Conclusion and Future Work

In this thesis, concepts for the flexible processing of streamed context data in a distributed environment have been presented. Different aspects have been highlighted and discussed. This chapter provides a conclusion in Section 8.1 and an outlook on future work in the remaining sections.

8.1 Conclusion

After the brief introduction and the enumeration of contributions in Chapter 1, in Chapter 2 motivating example scenarios from the domain of context-aware applications have been presented. These complex scenarios of context-aware applications were intended to extract requirements which a DSPS must meet in order to support context-aware applications adequately.

Based on the extracted requirements, in Chapter 3 an analysis of the existing state-of-the-art in distributed stream processing was presented. NexusDS is the DSPS which has been created during this thesis. Its architecture, its components, and its basic mode of operation was introduced in Chapter 3. NexusDS addresses the single requirements by its particular implemented solutions. Each of these solutions was introduced in its respective chapter.

In Chapter 4, the constraint model for NexusDS was introduced. Constraints naturally arise in the domain of context-aware applications. These applications heavily depend on custom functionality in terms of data processing techniques as well as interaction mechanisms. Furthermore, the particularities of the operator model and service model in NexusDS were introduced. The respective models target the extensibility of the NexusDS system towards custom functionality with a focus on the tight integration of these components. Finally, in this chapter the SP graph model of NexusDS and its (static) source data management was presented.

In Chapter 5, security aspects from the previous chapters were discussed. Protection goals have been defined which are important in the context of DSPS. A classification for access control and processing control mechanisms as well as a comparative evaluation of related concepts are provided. The security concept in NexusDS provides the augmentation of the original SP graph with security-relevant aspects regarding the access of data, its processing and the granularity in which data is accessed and processed. This augmentation is performed before SP graph deployment. Thus, an adequate deployment technique which takes into consideration the particular restrictions, i. e. security-relevant ones, deployment and runtime constraints of an operator, and QoS constraints are necessary. Last but not least a concept to evaluate these constraints and finding a suitable deployment for the given SP graph is mandatory.

In Chapter 6 an operator placement strategy called M-TOP was presented. M-TOP is a QoS-aware multi-target operator placement algorithm. M-TOP seeks for an operator placement that fulfills the specified QoS targets. The M-TOP heuristics aim at an early elimination of placements that do not lead to suitable solutions. Developers of SP graphs formulate their QoS targets according to application needs. The initial distribution has a big impact on the runtime behavior of a DSPS. An inappropriate initial SP graph distribution degrades execution and may lead to big overhead during runtime, e. g. by migrating operators with heavy state.

In order to ease the development process, tool support for such a complex and demanding environment is mandatory. On this behalf, the NexusDSEditor has been created which provides tool support for the development process of applications in NexusDS. It ranges from the modeling of context data to context data integration into the context management platform to the definition and deployment of SP graphs. Besides the tool support, two context-aware applications have been implemented to demonstrate the flexibility and the efficiency of NexusDS. The first application is a distributed visualization pipeline which displays the air flux in buildings via streamlines. Experiments have shown that the overhead introduced by the NexusDS framework is negligible. The second application scenario was a supervising tool in a manufacturing context called the Nexus Xplorer. The Nexus Xplorer aims at helping employees to find the reason for errors during production. It provides ways to visualize current production status as well as historical ones. By this, it is possible to introspect the status of all components involved at the time the error occurred.

8.2 Future Work

The presented thesis introduces great opportunities for future research. In the next sections, a brief sketch of potential future work directions are provided. The first one consists of optimizing the deployment of multiple SP graphs (Section 8.2.1). The second one considers adapting the execution of currently running SP graphs to changing conditions (Section 8.2.2). As a special action, the integration of application-specific adaptation logic could be interesting (Section 8.2.3). Finally, the NexusDSEditor could be further extended by integrating live monitoring as well as administrative tasks such as live operator migration into the editor (Section 8.2.4).

8.2.1 Deployment of Multiple SP-Graphs

The deployment of multiple SP graphs is an interesting future field of research. For this intermediate results might be shared among different SP graphs to reduce processing costs and thus increase scalability by finding common subexpressions as proposed by Sellis [122, 123]. Here, two different approaches are possible:

- optimizing the deployment of a SP graphs set, or
- optimizing the deployment of a SP graph sequence.

An SP graph set is represented by a number of SP graphs that arrive. This set has a fixed number of SP graphs and its joint execution is optimized. Contrarily, an SP graph sequence means an SP graph comes in at a time instant and is processed against all running SP graphs.

In the former case, the deployment algorithm has to cope with the optimal deployment of an SP graph set. In this case, the whole set of SP graphs which must be deployed is known before. Thus, its collective execution can be optimized. In the context of relational DBs there exists a proposal by Kraft [77]. The author proposes a coarse-grained optimization by applying heuristic rules to rewrite a given set of SQL statements. The rewriting is such that the common execution of these statements is optimized w.r.t. I/O and execution time. The application of the heuristic rules to the domain of DSPSs possibly offers high potential for further research in this direction.

When optimizing an SP graph sequence, each arriving SP graph that is going to be deployed and executed must be checked against all already running SP graphs. This is very challenging as the already running SP graphs should not be altered because this often results in heavy-weight adaptation. Thus, the newly arrived SP graph must best be integrated in the already running SP graphs. This might result in suboptimal choices or might actually require the adaptation of already running SP graphs.

In literature, many promising ideas can be found for multi SP graph deployment which might serve as a solid basis for future research. Dobra et al. [52] propose to split up continuous data streams into individual finite fragments. This allows to handle them as classical data in DBMSs. For this, query trees are constructed and scanned for equivalent branches, i. e. branches that compute the same intermediate results. As a consequence of the fragmentation process new problem areas emerge. Therefore, Chen et al. [36] propose an optimization approach which groups all streams using a common signature. In a second step, this allows to evaluate common subexpressions once. Chen et al. [35] proposed to check how to arrange operations such as *Select* and *Join* for grouped streams. Hong et al. [68] propose a framework that heuristically selects a suitable optimization strategy for specific applications. Seshadri et al. [124] presents an approach that is optimized for the processing of many simultaneous requests for data streams and distributed data sources. All these approaches represent a good starting point for future research. However, these approaches represent isolated solutions. An integrated view of the possibilities of optimizing the deployment of multiple SP graphs by considering adaptive data stream processing is missing.

8.2.2 Adapt SP-Graph Execution to Changing Conditions

The adaptation of currently running SP graphs to changing system conditions is a promising field. Two approaches could be interesting which have also been introduced in Section 3.5.4. The two adaptation principles are called lightweight adaptation and heavyweight adaptation. The former groups adaptation principles which are not demanding in terms of resources. Such a lightweight adaptation is e. g. the modification of the parametrization of a running operator. The latter adaptation principle refers to techniques which are usually demanding, such as the migration of an operator from one compute node to another. However, both adaptation principles apply to the time when the SP graph is already deployed, thus being co-called *post-deployment adaptation* steps. We could also imagine so-called *pre-deployment adaptation* steps which are applied before the SP graph is deployed and executed. In this case, the adaptation may consist in modifying the requirements an operator has, so that more resources are requested than needed by increasing the runtime constraints for necessary memory.

The adaptation mechanisms discussed all refer to adaptations on an SP graph level. However, we could also imagine the adaptation of the DSPS itself. In this case, instead of performing adaptation steps on the SP graph the system characteristics are modified. For this adaptation variant—thus during post-deployment adaptation—the adaptation could consist of assigning more resources to the currently executed SP graph on one node. In this case, the combination of both, i. e. pre-deployment and post-deployment adaptation mechanisms must be coordinated.

In the area of data stream processing, there already exist different approaches to achieve adaptivity approaches that aim to minimize latency and network load [6, 112]. Also, approaches which adapt in order to avoid overload situations and furthermore to adherence to previously defined quality measures [31] exist. Similarly, approaches have been investigated dealing with node failures [18, 69]. The adaptation might result in migration of certain parts of the SP graph, i. e. operators, from one compute node to another, for which dedicated migration techniques are mandatory [151, 155]. In addition to the adaptation of DSPSs, procedures selecting the initial configuration of the system are possible, so that it is robust against short-term changes. These configurations might reduce potentially costly adaptations, in particular the migration of operators [149]. The results of this thesis as well as the preliminary related work mentioned above form a good basis for the development of an integrated and comprehensive approach for adaptation.

8.2.3 Integration of Application-specific Adaptation Mechanisms

As well as system-side adaptation mechanisms, applications may also require adaptation opportunities. The full version of this adaptation mechanism at the application site is contrary to that, given there is only a limited view of the current state of the system. A transformation of the SP graph by the application is often inadequate as it potentially lacks system-side information and thus might end up in wrong conclusions. Therefore, it makes sense to integrate

application-defined adaptation logic into the SP graphs which are evaluated by the SP graph itself. The adaptations are controlled by adaptation indicators which are integrated into the SP graph at modeling time before execution. Adaptation indicators can be realized, e. g. as special operators or data sources that control the adaptation processes by their output data stream results. E. g., the adaptation process might be to change the selectivity of a filtering operator. This is useful if the postponed operators are not capable of processing the amount of data that the filtering operator forwards. [2] proposes to equip operators with special control connectors whose parameters are calculated as a function of its input data at runtime. This indeed allows to integrate runtime-dependent adaptation logic, but it is limited only to data which is exchanged between the operators. It does not provide the possibility to also integrate system-related state into its consideration.

8.2.4 Extending the NexusDSEditor by Dashboard Functionality

As presented in Chapter 7 the NexusDSEditor provides integrated tool support for the development of context data and context-aware and stream-based applications. To this end, mobile applications are in focus, as mobile devices typically provide only limited processing capabilities. For this reason, to outsource processing is mandatory. Therefore, modern DSPSs provide a suitable environment. They allow to distribute the single processing steps to suitable computers which carry out the processing tasks. In this regard, one of the questions is on how to distribute the SP graph to the different processing nodes. The NexusDSEditor already allows to create SP graphs and to define their distribution as well as their execution specifics for NexusDS. However, during execution problems may occur, making a rescheduling of the currently running SP graph mandatory. It is also helpful for the application developer as well as for the DSPS administrator to monitor the SP graph execution live and to find potential problems early in order to react to them.

Therefore, the NexusDSEditor could be extended towards live tracking of the environmental conditions, such as the current node statistics, and the live monitoring of executed SP graphs as future work. In order to allow live monitoring, appropriate data sinks for the NexusDSEditor must be implemented which are able to collect data first from the Monitoring Service (MS) to track the status of the single NexusDS nodes. Secondly, the operator model must be extended to reflect the possibility to transmit current runtime conditions to the NexusDSEditor. In turn this means for the NexusDSEditor that visualization plugins must be realized which display appropriately the measurement data. All this turns the NexusDSEditor into a dashboard.

Besides, the NexusDSEditor and the NexusDS system could be extended to implement the possibility to change the placement of SP graph fragments dynamically in order to test different configurations. Therefore, the NexusDS system must be extended towards management actions. These actions comprise the instantiation of additional services in NexusDS and the migration of currently executed SP graph fragments as described in [151].

List of Figures

List of Tables

Listings

Author Publications

- Nazario Cipriani, Daniela Nicklas, Matthias Großmann, Nicola Hönle, Carlos Lübbe, and Bernhard Mitschang. Verteilte Datenstromverarbeitung von Sensordaten. *Datenbank-Spektrum*, (28):37–42, Februar 2009.
- Nazario Cipriani, Matthias Wieland, Matthias Großmann, and Daniela Nicklas. Tool support for the design and management of context models. *Inf. Syst.*, 36(1):99–114, 2011.
- Nazario Cipriani, Oliver Dörler, and Bernhard Mitschang. Sicherer Zugriff und sichere Verarbeitung von Kontextdatenströmen in einer verteilten Umgebung. *Datenbank-Spektrum*, 12(1):13–22, 2012.
- Nazario Cipriani, Matthias Großmann, Daniela Nicklas, and Bernhard Mitschang. Federated Spatial Cursors. In Lúbia Vinhas and Antônio Carlos da Rocha Costa, editors, *GeoInfo*, pages 85–96. INPE, 2007.
- Nazario Cipriani, Matthias Wieland, Matthias Großmann, and Daniela Nicklas. Tool Support for the Design and Management of Spatial Context Models. In Janis Grundspenkis, Tadeusz Morzy, and Gottfried Vossen, editors, *ADBIS*, volume 5739 of *Lecture Notes in Computer Science*, pages 74–87. Springer, 2009.
- Nazario Cipriani, Mike Eissele, Andreas Brodt, Matthias Großmann, and Bernhard Mitschang. NexusDS: A Flexible and Extensible Middleware for Distributed Stream Processing. In Bipin C. Desai, Domenico Saccà, and Sergio Greco, editors, *IDEAS*, ACM International Conference Proceeding Series, pages 152–161. ACM, 2009.
- Nazario Cipriani, Matthias Großmann, Harald Sanftmann, and Bernhard Mitschang. Design Considerations of a Flexible Data Stream Processing Middleware. In Johann Eder, Mária Bieliková, and A Min Tjoa, editors, *ADBIS*, volume 789 of *CEUR Workshop Proceedings*, pages 222–231. CEUR-WS.org, 2011.
- Nazario Cipriani, Oliver Schiller, and Bernhard Mitschang. M-TOP: Mmulti-Target Operator Placement of Query Graphs for Data Streams. In Bipin C. Desai, Isabel F. Cruz, and Jorge Bernardino, editors, *IDEAS*, pages 52–60. ACM, 2011.

- Nazario Cipriani and Carlos Lübbe. Ausnutzung von Restriktionen zur Verbesserung des Deployment-Vorgangs des Verteilten Datenstromverarbeitungssystems NexusDS. In Stefan Fischer, Erik Maehle, and Rüdiger Reischuk, editors, *GI Jahrestagung*, volume 154 of *LNI*, pages 1985–1999. GI, 2009.
- Nazario Cipriani, Carlos Lübbe, and Alexander Moosbrugger. Exploiting Constraints to Build a Flexible and Extensible Data Stream Processing Middleware. In *IPDPS Workshops*, pages 1–8. IEEE, 2010.
- Nazario Cipriani, Carlos Lübbe, and Oliver Dörler. NexusDSEditor - Integrated Tool Support for the Data Stream Processing Middleware NexusDS. In Theo Härder, Wolfgang Lehner, Bernhard Mitschang, Harald Schöning, and Holger Schwarz, editors, *BTW*, volume 180 of *LNI*, pages 714–717. GI, 2011.
- Nazario Cipriani and Christoph Stach and Oliver Dörler and Bernhard Mitschang. NexusDSS: A System for Security Compliant Processing of Data Streams. In *DATA 2012: International Conference on Data Technologies and Applications. July 2012, Rome, Italy*, 2012. Winner of the best paper award.
- Andreas Brodt and Nazario Cipriani. NexusWeb - eine kontextbasierte Webanwendung im World Wide Space. In Johann Christoph Freytag, Thomas Ruf, Wolfgang Lehner, and Gottfried Vossen, editors, *BTW*, volume 144 of *LNI*, pages 588–591. GI, 2009.
- Ralph Lange, Nazario Cipriani, Lars Geiger, Matthias Großmann, Harald Weinschrott, Andreas Brodt, Matthias Wieland, Stamatia Rizou, and Kurt Rothermel. Making the World Wide Space Happen: New Challenges for the Nexus Context Platform. In *PerCom*, pages 1–4. IEEE Computer Society, 2009.
- Carlos Lübbe, Andreas Brodt, Nazario Cipriani, and Harald Sanftmann. NexusVIS: A Distributed Visualization Toolkit for Mobile Applications. In *PerCom Workshops*, pages 841–843. IEEE, 2010.
- Stamatia Rizou, Kai Häussermann, Frank Dürr, Nazario Cipriani, and Kurt Rothermel. A System for Distributed Context Reasoning. In *ICAS*, pages 84–89. IEEE Computer Society, 2010.
- Harald Sanftmann, Nazario Cipriani, and Daniel Weiskopf. Distributed context-aware visualization. In *PerCom Workshops*, pages 251–256. IEEE, 2011.
- Carlos Lübbe, Andreas Brodt, Nazario Cipriani, Matthias Großmann, and Bernhard Mitschang. DiSCO: A Distributed Semantic Cache Overlay for Location-Based Services. In Arkady B. Zaslavsky, Panos K. Chrysanthis, Dik Lun Lee, Dipanjan Chakraborty, Vana Kalogeraki, Mohamed F. Mokbel, and Chi-Yin Chow, editors, *Mobile Data Management (1)*, pages 17–26. IEEE, 2011.
- Carlos Lubbe and Nazario Cipriani. SimPl: A Simulation Platform for Elastic Load-Balancing in a Distributed Spatial Cache Overlay. In *Proceedings of the 2012 IEEE 13th International Conference on Mobile Data Management (mdm 2012)*, MDM '12, pages 340–343, Washington, DC, USA, 2012. IEEE Computer Society.

- Oliver Schiller, Nazario Cipriani, and Bernhard Mitschang. ProRea: Live Database Migration for Multi-tenant RDBMS with Snapshot Isolation. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 53–64, New York, NY, USA, 2013. ACM.

Bibliography

[1]

[2] Abadi, Daniel J. ; Ahmad, Yanif ; Balazinska, Magdalena ; Çetintemel, Ugur ; Cherniack, Mitch ; Hwang, Jeong-Hyon ; Lindner, Wolfgang ; Maskey, Anurag ; Rasin, Alex ; Ryvkina, Esther ; Tatbul, Nesime ; Xing, Ying ; Zdonik, Stanley B.: The Design of the Borealis Stream Processing Engine. In: *CIDR*, 2005, S. 277–289

[3] Abadi, Daniel J. ; Carney, Don ; Çetintemel, Ugur ; Cherniack, Mitch ; Convey, Christian ; Lee, Sangdon ; Stonebraker, Michael ; Tatbul, Nesime ; Zdonik, Stan: Aurora: a new model and architecture for data stream management. In: *The VLDB Journal* 12 (2003), Nr. 2, S. 120–139. – ISSN 1066–8888

[4] Abowd, Gregory D. ; Atkeson, Christopher G. ; Hong, Jason I. ; Long, Sue ; Kooper, Rob ; Pinkerton, Mike: Cyberguide: A mobile context-aware tour guide. In: *Wireless Networks* 3 (1997), Nr. 5, S. 421–433

[5] Ahmad, Yanif ; Berg, Bradley ; Çetintemel, Ugur ; Humphrey, Mark ; Hwang, Jeong-Hyon ; Jhingran, Anjali ; Maskey, Anurag ; Papaemmanouil, Olga ; Rasin, Alex ; Tatbul, Nesime ; Xing, Wenjuan ; Xing, Ying ; Zdonik, Stanley B.: Distributed operation in the Borealis stream processing engine. In: Özcan, Fatma (Hrsg.): *SIGMOD Conference*, ACM, 2005. – ISBN 1–59593–060–4, 882-884

[6] Ahmad, Yanif ; Çetintemel, Ugur: Networked Query Processing for Distributed Stream-Based Applications. In: Nascimento, Mario A. (Hrsg.) ; Özsu, M. T. (Hrsg.) ; Kossmann, Donald (Hrsg.) ; Miller, Renée J. (Hrsg.) ; Blakeley, José A. (Hrsg.) ; Schiefer, K. B. (Hrsg.): *VLDB*, Morgan Kaufmann, 2004. – ISBN 0–12–088469–0, 456-467

[7] Almeida, Victor T. ; Güting, Ralf H.: Supporting uncertainty in moving objects in network databases. In: Shahabi, Cyrus (Hrsg.) ; Boucelma, Omar (Hrsg.): *GIS*, ACM, 2005. – ISBN 1–59593–146–5, 31-40

[8] Alturi, Vijay ; Ferraiolo, David F.: Role-Based Access Control. In: Tilborg, Henk C. A. (Hrsg.) ; Jajodia, Sushil (Hrsg.): *Encyclopedia of Cryptography and Security (2nd*

Ed.). Springer, 2011. – ISBN 978–1–4419–5905–8, S. 1053–1055

[9] Amini, Lisa ; Andrade, Henrique ; Bhagwan, Ranjita ; Eskesen, Frank ; King, Richard ; Selo, Philippe ; Park, Yoonho ; Venkatramani, Chitra: SPC: a distributed, scalable platform for data mining. In: *DMSSP '06: Proceedings of the 4th international workshop on Data mining standards, services and platforms.* New York, NY, USA : ACM, 2006. – ISBN 1–59593–443–X, S. 27–37

[10] Amini, Lisa ; Jain, Navendu ; Sehgal, Anshul ; Silber, Jeremy ; Verscheure, Olivier: Adaptive Control of Extreme-scale Stream Processing Systems. In: *ICDCS*, IEEE Computer Society, 2006. – ISBN 0–7695–2540–7, 71

[11] Anderson, Ross J.: *Security engineering - a guide to building dependable distributed systems (2. ed.).* Wiley, 2008. – I–XL, 1–1040 S. – ISBN 978–0–470–06852–6

[12] Antoniu, Gabriel ; Hatcher, Philip J. ; Jan, Mathieu ; Noblet, David A.: *Performance evaluation of JXTA communication layers.* IEEE Computer Society, 2005. – 251–258 S.

[13] Appelrath, H.-Jürgen ; Geesen, Dennis ; Grawunder, Marco ; Michelsen, Timo ; Nicklas, Daniela: 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.* New York, NY, USA : ACM, 2012 (DEBS '12). – ISBN 978–1–4503–1315–5, 367–368

[14] Arasu, Arvind ; Babcock, Brian ; Babu, Shivnath ; Datar, Mayur ; Ito, Keith ; Motwani, Rajeev ; Nishizawa, Itaru ; Srivastava, Utkarsh ; Thomas, Dilys ; Varma, Rohit ; Widom, Jennifer: STREAM: The Stanford Stream Data Manager. In: *IEEE Data Eng. Bull.* 26 (2003), Nr. 1, S. 19–26

[15] Arasu, Arvind ; Babu, Shivnath ; Widom, Jennifer: CQL: A Language for Continuous Queries over Streams and Relations. In: Lausen, Georg (Hrsg.) ; Suciu, Dan (Hrsg.): *DBPL* Bd. 2921, Springer, 2003 (Lecture Notes in Computer Science). – ISBN 3–540–20896–8, 1-19

[16] Babcock, B. ; Babu, S. ; Datar, M. ; Motwani, R. ; Widom, J.: Models and issues in data stream systems. In: Kolaitis, Phokion G. (Hrsg.): *Proceedings of the 21nd Symposium on Principles of Database Systems*, ACM Press, 2002, S. 1–16

[17] Babcock, Brian ; Babu, Shivnath ; Datar, Mayur ; Motwani, Rajeev: Chain : Operator Scheduling for Memory Minimization in Data Stream Systems. In: Halevy, Alon Y. (Hrsg.) ; Ives, Zachary G. (Hrsg.) ; Doan, AnHai (Hrsg.): *SIGMOD Conference*, ACM, 2003. – ISBN 1–58113–634–X, 253-264

[18] Balazinska, Magdalena ; Balakrishnan, Hari ; Madden, Samuel ; Stonebraker, Michael: Fault-tolerance in the Borealis distributed stream processing system. In: Özcan, Fatma (Hrsg.): *SIGMOD Conference*, ACM, 2005. – ISBN 1–59593–060–4, 13-24

[19] Balazinska, Magdalena ; Balakrishnan, Hari ; Stonebraker, Michael: Load Management and High Availability in the Medusa Distributed Stream Processing System. In: Weikum, Gerhard (Hrsg.) ; König, Arnd C. (Hrsg.) ; Deßloch, Stefan (Hrsg.): *SIGMOD*

Conference, ACM, 2004. – ISBN 1–58113–859–8, 929-930

[20] Balke, Wolf-Tilo ; Güntzer, Ulrich: Multi-objective Query Processing for Database Systems. In: Nascimento, Mario A. (Hrsg.) ; Özsu, M. T. (Hrsg.) ; Kossmann, Donald (Hrsg.) ; Miller, Renée J. (Hrsg.) ; Blakeley, José A. (Hrsg.) ; Schiefer, K. B. (Hrsg.): *VLDB*, Morgan Kaufmann, 2004. – ISBN 0–12–088469–0, 936-947

[21] Ballard, Chuck ; Foster, Kevin ; Frenkiel, Andy ; Gedik, Bugra ; Koranda, Michael P. ; Nathan, Senthil ; Rajan, Deepak ; Rea, Roger ; Spicer, Mike ; Williams, Brian ; Zoubov, Vitali N.: *IBM InfoSphere Streams: Assembling Continuous Insight in the Information Revolution.* 2011 (IBM Redbook). – http://www.redbooks.ibm.com/abstracts/sg247970.html

[22] Bauer, M. ; Jendoubi, L. ; Siemoneit, O.: Smart Factory–Mobile Computing in Production Environments. In: *Proc. of the MobiSys 2004 Workshop on Applications of Mobile Embedded Systems (WAMES 04)* 3 (2004)

[23] Bertsimas, Dimitris ; Tsitsiklis, John: Simulated Annealing. In: *Statistical Science* 8 (1993), 02, Nr. 1, 10–15. **http://dx.doi.org/10.1214/ss/1177011077**. – DOI 10.1214/ss/1177011077

[24] Bettini, Claudio ; Brdiczka, Oliver ; Henricksen, Karen ; Indulska, Jadwiga ; Nicklas, Daniela ; Ranganathan, Anand ; Riboni, Daniele: A Survey of Context Modelling and Reasoning Techniques. In: *Pervasive and Mobile Computing* 6 (2010), apr, Nr. 2, S. 161–180. – ISSN 1574–1192

[25] Biem, Alain ; Bouillet, Eric ; Feng, Hanhua ; Ranganathan, Anand ; Riabov, Anton ; Verscheure, Olivier ; Koutsopoulos, Haris ; Moran, Carlos: IBM infosphere streams for scalable, real-time, intelligent transportation services. In: *Proceedings of the 2010 international conference on Management of data*. New York, NY, USA : ACM, 2010 (SIGMOD '10). – ISBN 978–1–4503–0032–2, 1093–1104

[26] Bolles, Andre ; Grawunder, Marco ; Jacobi, Jonas ; Nicklas, Daniela ; Appelrath, Hans-Jürgen: Odysseus: Ein Framework für massgeschneiderte Datenstrommanagementsystem. In: *GI Jahrestagung*, 2009, S. 2000–2014

[27] Brendan McGuigan: *How Big is the Internet?* http://www.wisegeek.com/how-big-is-the-internet.htm, October 2011. – Conjecture Corporation

[28] Cao, Hu ; Wolfson, Ouri ; Trajcevski, Goce: Spatio-temporal data reduction with deterministic error bounds. In: *VLDB J.* 15 (2006), Nr. 3, S. 211–228

[29] Cao, Jianneng ; Carminati, Barbara ; Ferrari, Elena ; Tan, Kian-Lee: ACStream: Enforcing Access Control over Data Streams. In: *ICDE*, IEEE, 2009. – ISBN 978–0–7695–3545–6, 1495-1498

[30] Carminati, Barbara ; Ferrari, Elena ; Tan, Kian-Lee: Specifying Access Control Policies on Data Streams. In: Ramamohanarao, Kotagiri (Hrsg.) ; Krishna, P. R. (Hrsg.) ; Mohania, Mukesh K. (Hrsg.) ; Nantajeewarawat, Ekawit (Hrsg.): *DASFAA* Bd. 4443, Springer, 2007 (Lecture Notes in Computer Science). – ISBN 978–3–540–71702–7, 410-

421

[31] Carney, Donald ; Çetintemel, Ugur ; Cherniack, Mitch ; Convey, Christian ; Lee, Sangdon ; Seidman, Greg ; Stonebraker, Michael ; Tatbul, Nesime ; Zdonik, Stanley B.: Monitoring Streams - A New Class of Data Management Applications. In: *VLDB*, Morgan Kaufmann, 2002, 215-226

[32] Carney, Donald ; Çetintemel, Ugur ; Rasin, Alex ; Zdonik, Stanley B. ; Cherniack, Mitch ; Stonebraker, Michael: Operator Scheduling in a Data Stream Manager. In: *VLDB*, 2003, 838-849

[33] Chappell, David A.: *Enterprise service bus - theory in practice.* O'Reilly, 2004. – I–XXIII, 1–247 S. – ISBN 978–0–596–00675–4

[34] Chaudhry, Nauman A. (Hrsg.) ; Shaw, Kevin (Hrsg.) ; Abdelguerfi, Mahdi (Hrsg.): *Advances in Database Systems*. Bd. 30: *Stream Data Management*. Springer, 2005. – ISBN 978–0–387–24393–1

[35] Chen, Jianjun ; DeWitt, David J. ; Naughton, Jeffrey F.: Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. In: Agrawal, Rakesh (Hrsg.) ; Dittrich, Klaus R. (Hrsg.): *ICDE*, IEEE Computer Society, 2002. – ISBN 0–7695–1531–2, 345-356

[36] Chen, Jianjun ; DeWitt, David J. ; Tian, Feng ; Wang, Yuan: NiagaraCQ: A Scalable Continuous Query System for Internet Databases. (2000), S. 379–390. ISBN 1–58113–218–2

[37] Cheng, Reynold ; Prabhakar, Sunil ; Kalashnikov, Dmitri V.: Querying Imprecise Data in Moving Object Environments. In: Dayal, Umeshwar (Hrsg.) ; Ramamritham, Krithi (Hrsg.) ; Vijayaraman, T. M. (Hrsg.): *ICDE*, IEEE Computer Society, 2003. – ISBN 0–7803–7665–X, 723-725

[38] Cherniack, M. ; Balakrishnan, H. ; Balazinska, M. ; Carney, D. ; Cetintemel, U. ; Xing, Y. ; Zdonik, S.: Scalable Distributed Stream Processing, 2003

[39] Cheverst, Keith ; Davies, Nigel ; Mitchell, Keith ; Friday, Adrian ; Efstratiou, Christos: Developing a context-aware electronic tourist guide: some issues and experiences. In: Turner, Thea (Hrsg.) ; Szwillus, Gerd (Hrsg.): *CHI*, ACM, 2000. – ISBN 1–58113–216–6, 17-24

[40] Cipriani, Nazario ; Eissele, Mike ; Brodt, Andreas ; Großmann, Matthias ; Mitschang, Bernhard: NexusDS: A Flexible and Extensible Middleware for Distributed Stream Processing. In: Desai, Bipin C. (Hrsg.) ; Saccà, Domenico (Hrsg.) ; Greco, Sergio (Hrsg.): *IDEAS*, ACM, 2009 (ACM International Conference Proceeding Series). – ISBN 978–1–60558–402–7, 152-161

[41] Cipriani, Nazario ; Großmann, Matthias ; Nicklas, Daniela ; Mitschang, Bernhard: Federated Spatial Cursors. In: Vinhas, Lúbia (Hrsg.) ; Rocha Costa, Antônio C. (Hrsg.): *GeoInfo*, INPE, 2007. – ISBN 978–85–17–00036–2, 85-96

[42] Cipriani, Nazario ; Großmann, Matthias ; Sanftmann, Harald ; Mitschang, Bernhard: Design Considerations of a Flexible Data Stream Processing Middleware. In: Eder, Johann (Hrsg.) ; Bieliková, Mária (Hrsg.) ; Tjoa, A M. (Hrsg.): *ADBIS* Bd. 789, CEUR-WS.org, 2011 (CEUR Workshop Proceedings), 222-231

[43] Cipriani, Nazario ; Lübbe, Carlos ; Moosbrugger, Alexander: Exploiting constraints to build a flexible and extensible data stream processing middleware. In: *IPDPS Workshops*, IEEE, 2010, 1-8

[44] Cipriani, Nazario ; Schiller, Oliver ; Mitschang, Bernhard: M-TOP: multi-target operator placement of query graphs for data streams. In: Desai, Bipin C. (Hrsg.) ; Cruz, Isabel F. (Hrsg.) ; Bernardino, Jorge (Hrsg.): *IDEAS*, ACM, 2011. – ISBN 978–1–4503–0627–0, 52-60

[45] Cipriani, Nazario ; Wieland, Matthias ; Großmann, Matthias ; Nicklas, Daniela: Tool support for the design and management of context models. In: *Inf. Syst.* 36 (2011), Nr. 1, S. 99–114

[46] Conner, W. S. ; Krishnamurthy, Lakshman ; Want, Roy: Making Everyday Life Easier Using Dense Sensor Networks. In: Abowd, Gregory D. (Hrsg.) ; Brumitt, Barry (Hrsg.) ; Shafer, Steven A. (Hrsg.): *Ubicomp* Bd. 2201, Springer, 2001 (Lecture Notes in Computer Science). – ISBN 3–540–42614–0, 49-55

[47] Constantinescu, Carmen ; Heinkel, Uwe ; Blond, Jan L. ; Schreiber, Stephan ; Mitschang, Bernhard ; Westkämper, Engelbert: Flexible Integration of Layout Planning and Adaptive Assembly Systems in Digital Enterprises. In: *Proceedings of the 38th CIRP International Seiminar on Manufacturing Systems (CIRP ISMS)*, CIRP, May 2005, S. 10–18

[48] Cugola, Gianpaolo ; Margara, Alessandro: Processing Flows of Information: From Data Stream to Complex Event Processing. In: *ACM Comput. Surv.* 44 (2012), Juni, Nr. 3, 15:1–15:62. **http://dx.doi.org/10.1145/2187671.2187677**. – DOI 10.1145/2187671.2187677. – ISSN 0360–0300

[49] Date, Chris J.: *An introduction to database systems (7. ed.).* Addison-Wesley-Longman, 2000. – I–XXII, 1–938 S. – ISBN 978–0–201–68419–3

[50] Department of Defence and Donald C. Latham: *Department Of Defense Standard Department Of Defense Trusted Computer System Evaluation Criteria.* 1985

[51] Dey, Anind K. ; Abowd, Gregory D.: Towards a better understanding of context and context-awareness. 1999. – Forschungsbericht. – 304–307 S.

[52] Dobra, Alin ; Garofalakis, Minos N. ; Gehrke, Johannes ; Rastogi, Rajeev: Multi-query optimization for sketch-based estimation. In: *Inf. Syst.* 34 (2009), Nr. 2, S. 209–230

[53] Forlizzi, Luca ; Güting, Ralf H. ; Nardelli, Enrico ; Schneider, Markus: A Data Model and Data Structures for Moving Objects Databases. In: Chen, Weidong (Hrsg.) ; Naughton, Jeffrey F. (Hrsg.) ; Bernstein, Philip A. (Hrsg.): *SIGMOD Conference*, ACM, 2000. – ISBN 1–58113–218–2, 319-330

[54] Foster, Ian ; Kesselman, Carl: *The Grid 2: Blueprint for a New Computing Infrastructure (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, 2003. – ISBN 1558609334

[55] Garcia-Molina, Hector ; Papakonstantinou, Yannis ; Quass, Dallan ; Rajaraman, Anand ; Sagiv, Yehoshua ; Ullman, Jeffrey ; Vassalos, Vasilis ; Widom, Jennifer: The TSIMMIS approach to mediation: Data models and languages. In: *Journal of Intelligent Information Systems* 8 (2004), Nr. 2, S. 117–132

[56] Gartner Inc.: *Gartner Says Solving 'Big Data' Challenge Involves More Than Just Managing Volumes of Data*. http://www.gartner.com/it/page.jsp?id=1731916, June 2011

[57] Gedik, Bugra ; Andrade, Henrique ; Wu, Kun-Lung ; Yu, Philip S. ; Doo, Myungcheol: SPADE: the system s declarative stream processing engine. In: Wang, Jason Tsong-Li (Hrsg.): *SIGMOD Conference*, ACM, 2008. – ISBN 978–1–60558–102–6, 1123-1134

[58] Gedik, Bugra ; Wu, Kun-Lung ; Yu, Philip S. ; Liu, Ling: Adaptive load shedding for windowed stream joins. In: Herzog, Otthein (Hrsg.) ; Schek, Hans-Jörg (Hrsg.) ; Fuhr, Norbert (Hrsg.) ; Chowdhury, Abdur (Hrsg.) ; Teiken, Wilfried (Hrsg.): *CIKM*, ACM, 2005. – ISBN 1–59593–140–6, 171-178

[59] Glover, Fred ; Laguna, Manuel: *Tabu Search*. Norwell, MA, USA : Kluwer Academic Publishers, 1997. – ISBN 079239965X

[60] Golab, Lukasz ; Özsu, M. T.: Issues in data stream management. In: *SIGMOD Rec.* 32 (2003), Nr. 2, S. 5–14. – ISSN 0163–5808

[61] Goodnight, Nolan ; 0003, Rui W. ; Humphreys, Greg: Computation on Programmable Graphics Hardware. In: *IEEE Computer Graphics and Applications* 25 (2005), Nr. 5, S. 12–15

[62] Großmann, Matthias ; Bauer, Martin ; Hönle, Nicola ; Käppeler, Uwe-Philipp ; Nicklas, Daniela ; Schwarz, Thomas: Efficiently Managing Context Information for Large-Scale Scenarios. In: *PerCom*, IEEE Computer Society, 2005. – ISBN 0–7695–2299–8, 331-340

[63] Großmann, Matthias ; Bauer, Martin ; Hönle, Nicola ; Käppeler, Uwe-Philipp ; Nicklas, Daniela ; Schwarz, Thomas: Efficiently Managing Context Information for Large-Scale Scenarios. In: *PerCom*, IEEE Computer Society, 2005. – ISBN 0–7695–2299–8, 331-340

[64] Haas, Laura M. ; Kossmann, Donald ; Ursu, Ioana: Loading a Cache with Query Results. In: Atkinson, Malcolm P. (Hrsg.) ; Orlowska, Maria E. (Hrsg.) ; Valduriez, Patrick (Hrsg.) ; Zdonik, Stanley B. (Hrsg.) ; Brodie, Michael L. (Hrsg.): *VLDB*, Morgan Kaufmann, 1999. – ISBN 1–55860–615–7, 351-362

[65] Haber, R. B. ; McNabb, D. A.: Visualization Idioms: A Conceptual Model for Scientific Visualization Systems. In: Schriver, B. (Hrsg.) ; Nielson, G. M. (Hrsg.) ; Rosenblum, L. J. (Hrsg.): *Visualization in Scientific Computing*. IEEE Computer Society Press, 1990, S. 74–93

[66] Halpin, T.: *Information Modeling and Relational Databases (From Conceptual Analysis to Logical Design).* Morgan Kauffman, 2001

[67] Henricksen, Karen ; Indulska, Jadwiga: A Software Engineering Framework for Context-Aware Pervasive Computing. In: *PerCom*, IEEE Computer Society, 2004. – ISBN 0–7695–2090–1, 77-86

[68] Hong, Mingsheng ; Riedewald, Mirek ; Koch, Christoph ; Gehrke, Johannes ; Demers, Alan J.: Rule-based multi-query optimization. In: Kersten, Martin L. (Hrsg.) ; Novikov, Boris (Hrsg.) ; Teubner, Jens (Hrsg.) ; Polutin, Vladimir (Hrsg.) ; Manegold, Stefan (Hrsg.): *EDBT* Bd. 360, ACM, 2009 (ACM International Conference Proceeding Series). – ISBN 978–1–60558–422–5, 120-131

[69] Hwang, Jeong-Hyon ; Balazinska, Magdalena ; Rasin, Alex ; Çetintemel, Ugur ; Stonebraker, Michael ; Zdonik, Stanley B.: High-Availability Algorithms for Distributed Stream Processing. In: Aberer, Karl (Hrsg.) ; Franklin, Michael J. (Hrsg.) ; Nishio, Shojiro (Hrsg.): *ICDE*, IEEE Computer Society, 2005. – ISBN 0–7695–2285–8, 779-790

[70] Hönle, Nicola ; Großmann, Matthias ; Nicklas, Daniela ; Mitschang, Bernhard: Preprocessing Position Data of Mobile Objects. In: Meng, Xiaofeng (Hrsg.) ; Lei, Hui (Hrsg.) ; Grumbach, Stéphane (Hrsg.) ; Leong, Hong V. (Hrsg.): *MDM*, IEEE, 2008, 41-48

[71] Hönle, Nicola ; Käppeler, Uwe-Philipp ; Nicklas, Daniela ; Schwarz, Thomas ; Großmann, Matthias: Benefits of Integrating Meta Data into a Context Model. In: *PerCom Workshops*, IEEE Computer Society, 2005. – ISBN 0–7695–2300–5, 25-29

[72] IBM InfoSphere Streams: *http://www-01.ibm.com/software/data/ infosphere/streams/*

[73] Josifovski, Vanja ; Schwarz, Peter M. ; Haas, Laura M. ; Lin, Eileen T.: Garlic: a new flavor of federated query processing for DB2. In: Franklin, Michael J. (Hrsg.) ; Moon, Bongki (Hrsg.) ; Ailamaki, Anastassia (Hrsg.): *SIGMOD Conference*, ACM, 2002. – ISBN 1–58113–497–5, 524-532

[74] Kemper, Alfons ; Eickler, André: *Datenbanksysteme: Eine Einführung*. 7. München : Oldenbourg, 2009. – ISBN 978–3–486–59018–0

[75] Kesselman, Carl ; Foster, Ian: *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1998. – ISBN 1558604758

[76] Kidd, Cory D. ; Orr, Robert ; Abowd, Gregory D. ; Atkeson, Christopher G. ; Essa, Irfan A. ; MacIntyre, Blair ; Mynatt, Elizabeth D. ; Starner, Thad ; Newstetter, Wendy: The Aware Home: A Living Laboratory for Ubiquitous Computing Research. In: Streitz, Norbert A. (Hrsg.) ; Siegel, Jane (Hrsg.) ; Hartkopf, Volker (Hrsg.) ; Konomi, Shin'ichi (Hrsg.): *CoBuild* Bd. 1670, Springer, 1999 (Lecture Notes in Computer Science). – ISBN 3–540–66596–X, 191-198

[77] Kraft, Tobias: *Optimization of query sequences*, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Dissertation, September 2009. **http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?**

id=DIS-2009-03&engl=0. – 217 S.

[78] Krishnamurthy, Sailesh ; Chandrasekaran, Sirish ; Cooper, Owen ; Deshpande, Amol ; Franklin, Michael J. ; Hellerstein, Joseph M. ; Hong, Wei ; Madden, Samuel ; Reiss, Frederick ; Shah, Mehul A.: TelegraphCQ: An Architectural Status Report. In: *IEEE Data Eng. Bull.* 26 (2003), Nr. 1, S. 11–18

[79] Krämer, Jürgen ; Seeger, Bernhard: PIPES - A Public Infrastructure for Processing and Exploring Streams. In: Weikum, Gerhard (Hrsg.) ; König, Arnd C. (Hrsg.) ; Deßloch, Stefan (Hrsg.): *SIGMOD Conference*, ACM, 2004. – ISBN 1–58113–859–8, 925-926

[80] Kuntschke, Richard ; Stegmaier, Bernhard ; Kemper, Alfons ; Reiser, Angelika: StreamGlobe: Processing and Sharing Data Streams in Grid-Based P2P Infrastructures. In: Böhm, Klemens (Hrsg.) ; Jensen, Christian S. (Hrsg.) ; Haas, Laura M. (Hrsg.) ; Kersten, Martin L. (Hrsg.) ; Larson, Per-Åke (Hrsg.) ; Ooi, Beng C. (Hrsg.): *VLDB*, ACM, 2005. – ISBN 1–59593–177–5, 1259-1262

[81] Lakshmanan, Geetika T. ; Li, Ying ; Strom, Robert E.: Placement Strategies for Internet-Scale Data Stream Systems. In: *IEEE Internet Computing* 12 (2008), Nr. 6, S. 50–60

[82] Lakshmanan, Geetika T. ; Strom, Robert E.: Biologically-Inspired Distributed Middleware Management for Stream Processing Systems. In: Issarny, Valérie (Hrsg.) ; Schantz, Richard E. (Hrsg.): *Middleware* Bd. 5346, Springer, 2008 (Lecture Notes in Computer Science). – ISBN 978–3–540–89855–9, 223-242

[83] Lange, Ralph: *Scalable Management of Trajectories and Context Model Descriptions*, Universität Stuttgart : Sonderforschungsbereich SFB 627 (Nexus: Umgebungsmodelle für mobile kontextbezogene Systeme), Germany, Dissertation, Dezember 2010. **http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIS-2010-04&engl=0**. – 202 S.

[84] Lange, Ralph ; Cipriani, Nazario ; Geiger, Lars ; Großmann, Matthias ; Weinschrott, Harald ; Brodt, Andreas ; Wieland, Matthias ; Rizou, Stamatia ; Rothermel, Kurt: Making the World Wide Space Happen: New Challenges for the Nexus Context Platform. In: *PerCom*, IEEE Computer Society, 2009. – ISBN 978–1–4244–3304–9, 1-4

[85] Leonhardi, Alexander ; Kubach, Uwe ; Rothermel, Kurt ; Fritz, Andreas: Virtual Information Towers-A Metaphor for Intuitive, Location-Aware Information Access in a Mobile Environment. In: *ISWC*, IEEE Computer Society, 1999. – ISBN 0–7695–0428–0, 15-20

[86] Levy, Alon Y. ; Rajaraman, Anand ; Ordille, Joann J.: Querying Heterogeneous Information Sources Using Source Descriptions. In: Vijayaraman, T. M. (Hrsg.) ; Buchmann, Alejandro P. (Hrsg.) ; Mohan, C. (Hrsg.) ; Sarda, Nandlal L. (Hrsg.): *VLDB*, Morgan Kaufmann, 1996. – ISBN 1–55860–382–4, 251-262

[87] Levy, Alon Y. ; Srivastava, Divesh ; Kirk, Thomas: Data Model and Query Evaluation in Global Information Systems. In: *J. Intell. Inf. Syst.* 5 (1995), Nr. 2, S. 121–143

[88] Lindner, Wolfgang ; Meier, Jörg: Towards a Secure Data Stream Management System. In: Draheim, Dirk (Hrsg.) ; Weber, Gerald (Hrsg.): *TEAA* Bd. 3888, Springer, 2005 (Lecture Notes in Computer Science). – ISBN 3–540–32734–7, 114-128

[89] Lindner, Wolfgang ; Meier, Jörg: Securing the Borealis Data Stream Engine. In: *IDEAS*, IEEE Computer Society, 2006, 137-147

[90] Linthicum, David S.: *Enterprise application integration*. 3. printing. Addison-Wesley, 2000 (Addison-Wesley information technology series). – XXI, 377 S.. – ISBN 0–201–61583–5

[91] Lucke, Dominik ; Constantinescu, Carmen ; Westkämper, Engelbert: Smart Factory - A Step towards the Next Generation of Manufacturing. In: Mitsuishi, Mamoru (Hrsg.) ; Ueda, Kanji (Hrsg.) ; Kimura, Fumihiko (Hrsg.): *Manufacturing Systems and Technologies for the New Frontier*. Springer London, 2008. – ISBN 978–1–84800–267–8

[92] Luckham, David C.: *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2001. – ISBN 0201727897

[93] Mayur, Brian B. ; Babcock, Brian ; Datar, Mayur ; Motwani, Rajeev: Load Shedding Techniques for Data Stream Systems. In: *In Proc. of the 2003 Workshop on Management and Processing of Data Streams (MPDS*, 2003

[94] Merkle, Ralph C.: A Certified Digital Signature. In: Brassard, Gilles (Hrsg.): *CRYPTO* Bd. 435, Springer, 1989 (Lecture Notes in Computer Science). – ISBN 3–540–97317–6, 218-238

[95] Mitchell, M.: *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1998

[96] Mokbel, Mohamed F. ; Xiong, Xiaopeng ; Aref, Walid G. ; Hambrusch, Susanne E. ; Prabhakar, Sunil ; Hammad, Moustafa A.: PLACE: A Query Processor for Handling Real-time Spatio-temporal Data Streams. In: Nascimento, Mario A. (Hrsg.) ; Özsu, M. T. (Hrsg.) ; Kossmann, Donald (Hrsg.) ; Miller, Renée J. (Hrsg.) ; Blakeley, José A. (Hrsg.) ; Schiefer, K. B. (Hrsg.): *VLDB*, Morgan Kaufmann, 2004. – ISBN 0–12–088469–0, 1377-1380

[97] Mínguez, Jorge ; Ruthardt, Frank ; Riffelmacher, Philipp ; Scheibler, Thorsten ; Mitschang, Bernhard: Service-Based Integration in Event-Driven Manufacturing Environments. In: Chiu, Dickson K. W. (Hrsg.) ; Bellatreche, Ladjel (Hrsg.) ; Sasaki, Hideyasu (Hrsg.) ; Leung, Ho fung (Hrsg.) ; Cheung, Shing-Chi (Hrsg.) ; Hu, Haiyang (Hrsg.) ; Shao, Jie (Hrsg.): *WISE Workshops* Bd. 6724, Springer, 2010 (Lecture Notes in Computer Science). – ISBN 978–3–642–24395–0, 295-308

[98] Nehme, Rimma V. ; Lim, Hyo-Sang ; Bertino, Elisa: FENCE: Continuous access control enforcement in dynamic data stream environments. In: Li, Feifei (Hrsg.) ; Moro, Mirella M. (Hrsg.) ; Ghandeharizadeh, Shahram (Hrsg.) ; Haritsa, Jayant R. (Hrsg.) ; Weikum, Gerhard (Hrsg.) ; Carey, Michael J. (Hrsg.) ; Casati, Fabio (Hrsg.) ; Chang, Edward Y. (Hrsg.) ; Manolescu, Ioana (Hrsg.) ; Mehrotra, Sharad (Hrsg.) ; Dayal,

Umeshwar (Hrsg.) ; Tsotras, Vassilis J. (Hrsg.): *ICDE*, IEEE, 2010. – ISBN 978–1–4244–5444–0, 940-943

[99] Nehme, Rimma V. ; Rundensteiner, Elke A. ; Bertino, Elisa: A Security Punctuation Framework for Enforcing Access Control on Streaming Data. In: *ICDE*, IEEE, 2008, 406-415

[100] Ng, Kenneth W. ; Wang, Zhenghao ; Muntz, Richard R. ; Shek, Eddie C.: On Reconfiguring Query Execution Plans in Distributed Object-Relational DBMS. In: *ICPADS*, 1998, 59-66

[101] Nicklas, Daniela: *Ein umfassendes Umgebungsmodell als Integrationsstrategie für ortsbezogene Daten und Dienste*, Universität Stuttgart : Sonderforschungsbereich SFB 627 (Nexus: Umgebungsmodelle für mobile kontextbezogene Systeme), Germany, Dissertation, Februar 2006. **http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIS-2006-02&engl=0**. – 232 S.

[102] Nicklas, Daniela ; Großmann, Matthias ; Schwarz, Thomas: NexusScout: An Advanced Location-Based Application on a Distributed, Open Mediation Platform. In: *VLDB*, 2003, 1089-1092

[103] Nicklas, Daniela ; Großmann, Matthias ; Schwarz, Thomas ; Volz, Steffen ; Mitschang, Bernhard: A Model-Based, Open Architecture for Mobile, Spatially Aware Applications. In: Jensen, Christian S. (Hrsg.) ; Schneider, Markus (Hrsg.) ; Seeger, Bernhard (Hrsg.) ; Tsotras, Vassilis J. (Hrsg.): *SSTD* Bd. 2121, Springer, 2001 (Lecture Notes in Computer Science). – ISBN 3–540–42301–X, 117-135

[104] Nicklas, Daniela ; Hönle, Nicola ; Moltenbrey, Michael ; Mitschang, Bernhard: Design and Implementation Issues for Explorative Location-based Applications: The NexusRallye. In: Iochpe, Cirano (Hrsg.) ; Câmara, Gilberto (Hrsg.): *GeoInfo*, INPE, 2004. – ISBN 3–901882–20–0, 167-181

[105] Nicklas, Daniela ; Mitschang, Bernhard: On building location aware applications using an open platform based on the NEXUS Augmented World Model. In: *Software and System Modeling* 3 (2004), Nr. 4, S. 303–313

[106] Nicklas, Daniela ; Neumann, Carsten: NexusEditor: A Schema-Aware Graphical User Interface for Managing Spatial Context Models. In: Meng, Xiaofeng (Hrsg.) ; Lei, Hui (Hrsg.) ; Grumbach, Stéphane (Hrsg.) ; Leong, Hong V. (Hrsg.): *MDM*, IEEE, 2008, 213-214

[107] Owens, John D. ; Luebke, David ; Govindaraju, Naga ; Harris, Mark ; Krger, Jens ; Lefohn, Aaron E. ; Purcell, Timothy J.: A Survey of General-Purpose Computation on Graphics Hardware. In: *Eurographics 2005, State of the Art Reports*, 2005, S. 21–51

[108] Papadimitriou, Christos H. ; Yannakakis, Mihalis: Multiobjective Query Optimization. In: Buneman, Peter (Hrsg.): *PODS*, ACM, 2001. – ISBN 1–58113–361–8

[109] Papakonstantinou, Yannis ; Garcia-Molina, Hector ; Ullman, Jeffrey D.: MedMaker: A Mediation System Based on Declarative Specifications. In: Su, Stanley Y. W. (Hrsg.):

ICDE, IEEE Computer Society, 1996. – ISBN 0–8186–7240–4, 132-141

[110] Patroumpas, Kostas ; Sellis, Timos: Window Specification over Data Streams. In: *Current Trends in Database — Technology EDBT 2006* Bd. 4254. Springer Berlin / Heidelberg, 2006. – ISBN 978–3–540–46788–5, S. 445–464

[111] Pfoser, Dieter ; Jensen, Christian S.: Capturing the Uncertainty of Moving-Object Representations. In: Güting, Ralf H. (Hrsg.) ; Papadias, Dimitris (Hrsg.) ; Lochovsky, Frederick H. (Hrsg.): *SSD* Bd. 1651, Springer, 1999 (Lecture Notes in Computer Science). – ISBN 3–540–66247–2, 111-132

[112] Pietzuch, Peter R. ; Ledlie, Jonathan ; Shneidman, Jeffrey ; Roussopoulos, Mema ; Welsh, Matt ; Seltzer, Margo I.: Network-Aware Operator Placement for Stream-Processing Systems. In: Liu, Ling (Hrsg.) ; Reuter, Andreas (Hrsg.) ; Whang, Kyu-Young (Hrsg.) ; Zhang, Jianjun (Hrsg.): *ICDE*, IEEE Computer Society, 2006, 49

[113] Rizou, Stamatia ; Dürr, Frank ; Rothermel, Kurt: Providing QoS Guarantees in Large-Scale Operator Networks. In: *HPCC*, IEEE, 2010, 337-345

[114] Rizou, Stamatia ; Dürr, Frank ; Rothermel, Kurt: Solving the Multi-Operator Placement Problem in Large-Scale Operator Networks. In: *ICCCN*, IEEE, 2010. – ISBN 978–1–4244–7115–7, 1-6

[115] RTM Realtime Monitoring GmbH: *http://www.realtime-monitoring.de/*

[116] Rundensteiner, Elke A. ; Ding, Luping ; Sutherland, Timothy M. ; Zhu, Yali ; Pielech, Bradford ; Mehta, Nishant: CAPE: Continuous Query Engine with Heterogeneous-Grained Adaptivity. In: Nascimento, Mario A. (Hrsg.) ; Özsu, M. T. (Hrsg.) ; Kossmann, Donald (Hrsg.) ; Miller, Renée J. (Hrsg.) ; Blakeley, José A. (Hrsg.) ; Schiefer, K. B. (Hrsg.): *VLDB*, Morgan Kaufmann, 2004. – ISBN 0–12–088469–0, 1353-1356

[117] Sanftmann, Harald ; Cipriani, Nazario ; Weiskopf, Daniel: Distributed context-aware visualization. In: *PerCom Workshops*, IEEE, 2011, 251-256

[118] Schilit, Bill ; Adams, Norman ; Want, Roy: Context-Aware Computing Applications. In: *IEEE Workshop on Mobile Computing Systems and Applications.* Santa Cruz, CA, US, 1994

[119] Schollmeier, Rüdiger: A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications. (2001), S. 101–102. ISBN 0–7695–1503–7

[120] Schulte, Roy W. ; Natis, Yefim V.: "Service Oriented" Architectures, Part 1. (1996), 2. **http://www.gartner.com/DisplayDocument?id=302868**

[121] Schwarz, Thomas: *Verarbeitung ortsbezogener Anfragen in lose gekoppelten, föderierten Systemen: Konzeption, Realisierung, Bewertung.*, University of Stuttgart, Diss., 2007. **http://elib.uni-stuttgart.de/opus/volltexte/2007/3321/**. – http://d-nb.info/986743283

[122] Sellis, Timos K.: Global Query Optimization. (1986), S. 191–205

[123] Sellis, Timos K.: Multiple-Query Optimization. In: *ACM Trans. Database Syst.* 13 (1988), Nr. 1, S. 23–52

[124] Seshadri, Sangeetha ; Kumar, Vibhore ; Cooper, Brian F.: Optimizing Multiple Queries in Distributed Data Stream Systems. In: Barga, Roger S. (Hrsg.) ; Zhou, Xiaofang (Hrsg.): *ICDE Workshops*, IEEE Computer Society, 2006, 25

[125] Site, Nexus P.: *The Nexus Project Site.* **http://www.nexus.uni-stuttgart.de/**

[126] Srivastava, Utkarsh ; Munagala, Kamesh ; Widom, Jennifer: Operator placement for in-network stream query processing. In: Li, Chen (Hrsg.): *PODS*, ACM, 2005. – ISBN 1–59593–062–0, 250-258

[127] Stegmaier, Bernhard ; Kuntschke, Richard ; Kemper, Alfons: StreamGlobe: adaptive query processing and optimization in streaming P2P environments. In: Labrinidis, Alexandros (Hrsg.) ; Madden, Samuel (Hrsg.): *DMSN* Bd. 72, ACM, 2004 (ACM International Conference Proceeding Series), 88-97

[128] StreamBase Systems, Inc.: *http://www.streambase.com/*

[129] Sutherland, Timothy M. ; 0005, Bin L. ; Jbantova, Mariana ; Rundensteiner, Elke A.: D-CAPE: distributed and self-tuned continuous query processing. In: Herzog, Otthein (Hrsg.) ; Schek, Hans-Jörg (Hrsg.) ; Fuhr, Norbert (Hrsg.) ; Chowdhury, Abdur (Hrsg.) ; Teiken, Wilfried (Hrsg.): *CIKM*, ACM, 2005. – ISBN 1–59593–140–6, 217-218

[130] Sutherland, Timothy M. ; Zhu, Yali ; Ding, Luping ; Rundensteiner, Elke A.: An Adaptive Multi-Objective Scheduling Selection Framework for Continuous Query Processing. In: *IDEAS*, IEEE Computer Society, 2005. – ISBN 0–7695–2404–4, 445-454

[131] Tatbul, Nesime ; Zdonik, Stanley B.: Dealing with Overload in Distributed Stream Processing Systems. In: Barga, Roger S. (Hrsg.) ; Zhou, Xiaofang (Hrsg.): *ICDE Workshops*, IEEE Computer Society, 2006, 24

[132] Tatbul, Nesime ; Çetintemel, Ugur ; Zdonik, Stanley B. ; Cherniack, Mitch ; Stonebraker, Michael: Load Shedding in a Data Stream Manager. In: *VLDB*, 2003, 309-320

[133] Terry, Douglas B. ; Goldberg, David ; Nichols, David ; Oki, Brian M.: Continuous Queries over Append-Only Databases. (1992), S. 321–330

[134] Teubner, Jens ; Woods, Louis: Snowfall: Hardware Stream Analysis Made Easy. In: Härder, Theo (Hrsg.) ; Lehner, Wolfgang (Hrsg.) ; Mitschang, Bernhard (Hrsg.) ; Schöning, Harald (Hrsg.) ; Schwarz, Holger (Hrsg.): *BTW* Bd. 180, GI, 2011 (LNI). – ISBN 978–3–88579–274–1, 738-741

[135] The Korn/Ferry Institute: *Big Data Effect.* http://kornferrybriefings.com/latest_thinking/the_big_data_effect.php, 2011

[136] Tomasic, Anthony ; Raschid, Louiqa ; Valduriez, Patrick: Scaling Heterogeneous Databases and the Design of Disco. In: *ICDCS*, 1996, 449-457

[137] Trajcevski, Goce ; Cao, Hu ; Scheuermann, Peter ; Wolfson, Ouri ; Vaccaro, Dennis: On-line data reduction and the quality of history in moving objects databases. In:

Chrysanthis, Panos K. (Hrsg.) ; Jensen, Christian S. (Hrsg.) ; Kumar, Vijay (Hrsg.) ; Labrinidis, Alexandros (Hrsg.): *MobiDE*, ACM, 2006. – ISBN 1–59593–436–7, 19-26

[138] Trajcevski, Goce ; Wolfson, Ouri ; Zhang, Fengli ; Chamberlain, Sam: The Geometry of Uncertainty in Moving Objects Databases. In: Jensen, Christian S. (Hrsg.) ; Jeffery, Keith G. (Hrsg.) ; Pokorný, Jaroslav (Hrsg.) ; Saltenis, Simonas (Hrsg.) ; Bertino, Elisa (Hrsg.) ; Böhm, Klemens (Hrsg.) ; Jarke, Matthias (Hrsg.): *EDBT* Bd. 2287, Springer, 2002 (Lecture Notes in Computer Science). – ISBN 3–540–43324–4, 233-250

[139] Tryfona, Nectaria ; Jensen, Christian S.: Conceptual Data Modeling for Spatiotemporal Applications. In: *GeoInformatica* 3 (1999), Nr. 3, S. 245–268

[140] Tu, Yi-Cheng ; Liu, Song ; Prabhakar, Sunil ; Yao, Bin: Load Shedding in Stream Databases: A Control-Based Approach. In: Dayal, Umeshwar (Hrsg.) ; Whang, Kyu-Young (Hrsg.) ; Lomet, David B. (Hrsg.) ; Alonso, Gustavo (Hrsg.) ; Lohman, Guy M. (Hrsg.) ; Kersten, Martin L. (Hrsg.) ; Cha, Sang K. (Hrsg.) ; Kim, Young-Kuk (Hrsg.): *VLDB*, ACM, 2006. – ISBN 1–59593–385–9, 787-798

[141] Tucker, Peter A. ; Maier, David ; Sheard, Tim ; Fegaras, Leonidas: Exploiting Punctuation Semantics in Continuous Data Streams. In: *IEEE Trans. Knowl. Data Eng.* 15 (2003), Nr. 3, S. 555–568

[142] Urhan, Tolga ; Franklin, Michael J.: Dynamic Pipeline Scheduling for Improving Interactive Query Performance. In: Apers, Peter M. G. (Hrsg.) ; Atzeni, Paolo (Hrsg.) ; Ceri, Stefano (Hrsg.) ; Paraboschi, Stefano (Hrsg.) ; Ramamohanarao, Kotagiri (Hrsg.) ; Snodgrass, Richard T. (Hrsg.): *VLDB*, Morgan Kaufmann, 2001. – ISBN 1–55860–804–4, 501-510

[143] Vogelgesang, Thomas ; Geesen, Dennis ; Grawunder, Marco ; Nicklas, Daniela ; Appelrath, Hans-Jürgen: Scheduling von Datenströmen auf der Basis von Service Level Agreements. In: *Datenbank-Spektrum* 12 (2012), Nr. 1, 23-32. **http://dblp.uni-trier.de/db/journals/dbsk/dbsk12.html#VogelgesangGGNA12**

[144] Volz, S.: An Iterative Approach for Matching Multiple Representations of Street Data. In: *Proc. of the JOINT ISPRS Workshop on Multiple Representations and Interoperability of Spatial Data* Bd. XXXVI Part 2/W40, 2006

[145] Volz, S. ; Walter, V.: Linking different Geospatial Databases by explicit Relations. In: *Proceedings of the XXth Conference of ISPRS '04*, 2004

[146] Westkaemper, E. ; Jendoubi, L. ; Eissele, M. ; Ertl, T.: Smart Factory - Bridging the gap between digital planning and reality. In: *Proc. of the 38th CIRP Intl. Seminar on Manufacturing Systems*, CIRP, 2005

[147] Wieland, Matthias ; Nicklas, Daniela ; Leymann, Frank: Managing Technical Processes Using Smart Workflows. In: Mähönen, Petri (Hrsg.) ; Pohl, Klaus (Hrsg.) ; Priol, Thierry (Hrsg.): *ServiceWave* Bd. 5377, Springer, 2008 (Lecture Notes in Computer Science). – ISBN 978–3–540–89896–2, 287-298

[148] Wolfson, Ouri ; Xu, Bo ; Chamberlain, Sam ; Jiang, Liqin: Moving Objects Databases: Issues and Solutions. In: Rafanelli, Maurizio (Hrsg.) ; Jarke, Matthias (Hrsg.): *SSDBM*, IEEE Computer Society, 1998. – ISBN 0–8186–8575–1, 111-122

[149] Xing, Ying ; Hwang, Jeong-Hyon ; Çetintemel, Ugur ; Zdonik, Stanley B.: Providing Resiliency to Load Variations in Distributed Stream Processing. In: Dayal, Umeshwar (Hrsg.) ; Whang, Kyu-Young (Hrsg.) ; Lomet, David B. (Hrsg.) ; Alonso, Gustavo (Hrsg.) ; Lohman, Guy M. (Hrsg.) ; Kersten, Martin L. (Hrsg.) ; Cha, Sang K. (Hrsg.) ; Kim, Young-Kuk (Hrsg.): *VLDB*, ACM, 2006. – ISBN 1–59593–385–9, 775-786

[150] Xiong, Xiaopeng ; Elmongui, Hicham G. ; Chai, Xiaoyong ; Aref, Walid G.: Place: A Distributed Spatio-Temporal Data Stream Management System for Moving Objects. In: Becker, Christian (Hrsg.) ; Jensen, Christian S. (Hrsg.) ; Su, Jianwen (Hrsg.): *MDM*, IEEE, 2007, 44-51

[151] Yang, Yin ; Krämer, Jürgen ; Papadias, Dimitris ; Seeger, Bernhard: HybMig: A Hybrid Approach to Dynamic Plan Migration for Continuous Queries. In: *IEEE Trans. Knowl. Data Eng.* 19 (2007), Nr. 3, S. 398–411

[152] Zdonik, Stanley B. ; Stonebraker, Michael ; Cherniack, Mitch ; Çetintemel, Ugur ; Balazinska, Magdalena ; Balakrishnan, Hari: *The Aurora and Medusa Projects.* 2003

[153] Zhou, Yongluan ; Ooi, Beng C. ; Tan, Kian-Lee ; Wu, Ji: Efficient Dynamic Operator Placement in a Locally Distributed Continuous Query System. In: Meersman, Robert (Hrsg.) ; Tari, Zahir (Hrsg.): *OTM Conferences (1)* Bd. 4275, Springer, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–48287–3, 54-71

[154] Zhu, Mengxia ; Wu, Qishi ; Rao, N.S.V. ; Iyengar, S.: Adaptive visualization pipeline decomposition and mapping onto computer networks. In: *Image and Graphics, 2004. Proceedings. Third International Conference on* (2004), Dec., S. 402–405

[155] Zhu, Yali ; Rundensteiner, Elke A. ; Heineman, George T.: Dynamic Plan Migration for Continuous Queries Over Data Streams. In: Weikum, Gerhard (Hrsg.) ; König, Arnd C. (Hrsg.) ; Deßloch, Stefan (Hrsg.): *SIGMOD Conference*, ACM, 2004. – ISBN 1–58113–859–8, 431-442

[156] Zhu, Yunyue ; Shasha, Dennis: StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In: *VLDB*, Morgan Kaufmann, 2002, 358-369

Curriculum Vitæ

Nazario Cipriani	
Date of birth:	September 14th, 1978
Place of birth:	San Giovanni (FG), Italy
Nationality:	Italian
09/1985 – 07/1989	Primary School at Mörikeschule in Köngen, Germany
09/1989 – 07/1995	Secondary School at Burgschule Realschule in Köngen, Germany Degree: Mittlere Reife
09/1995 – 07/1998	Secondary School at Max-Eyth-Schule Gymnasium in Kirchheim, Germany Degree: Abitur
10/1998 – 11/2005	Studies in Computer Science at Universität Stuttgart, Germany Degree: Diplom-Informatiker (Dipl.-Inf.)
05/2005 – 11/2005	Diploma thesis at Universität Stuttgart in Stuttgart, Germany, Topic: “Cursor Concepts for the Nexus System”
01/2006 – 08/2012	Research staff member at the Institute of Parallel and Distributed Systems (IPVS), Universität Stuttgart, Germany