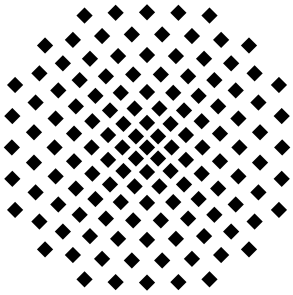

Universität Stuttgart
SFB 627 – NEXUS

Information Management and Exchange in the Nexus Platform

Version 2.0 – 30.4.04

Report Number 2004/04

Martin Bauer .....IPVS (VS)
Frank Dürr.....IPVS (VS)
Jan Geiger .....IPVS (VS)
Matthias Großmann .....IPVS (AS)
Nicola Hönle .....IPVS (AS)
Jean Joswig .....IPVS (VS)
Daniela Nicklas.....IPVS (AS)
Thomas Schwarz.....IPVS (AS)

Content

1 Introduction

1.1 About this document

This document is a technical report of the interdisciplinary research project Nexus which is a Center of Excellence that is carried out at the University of Stuttgart.

It is targeted at all interested readers who wish to delve deeper into the interfaces of the Nexus platform and the underlying architecture to get a better understanding of how the Nexus platform models the surrounding world and its manyfold context parameters.

This document is the successor of the Final Report of Design Workshop [Nicklas et al 2000], which at the time of its creation represented the specification for the implementation of the Nexus platform and went hand in hand with the formation of the research group Nexus. The successful research work accomplished in the following three years until 2003 uncovered even more open questions which resulted in the decision to found an interdisciplinary Center of Excellence. This document defines the specification of the Nexus platform's interfaces, so that all participating projects have a common starting point and are able to build interoperable components. It does not contain real protocol definitions yet. Especially it does not cover error cases. These details can partially be found in the specifications of implemented components; however they should be included in a future version of this document.

1.2 Introduction

Modeling real-world context demands the aggregation of a great number of parameters. These parameters are typically incorporated into small models of the environment and the combination of several of these models leads to the creation of a digital world model.

The following document wants to demonstrate how the Nexus project tries to manage this complex task and models the real world augmented by virtual entities.

After the introduction we first describe the platform design and give the reader an in-depth overview of the Nexus platform (Chapter 2). The following chapter explains the Nexus Standard Attribute Types (Chapter 3), which form the basic data types for the Nexus Augmented World. The Nexus Standard Attribute Schema (NSAS) in Chapter 4 defines the Nexus attributes which are used to build Nexus objects. Each attribute has to have one of the data types introduced in Chapter 3. The types (or classes) of these objects are defined in the Nexus Standard Class Schema (NSCS) which is described in Chapter 5. Each class comprises of a subset of the attributes defined in the NSAS and inherits from one or more base classes.

The Augmented World Modeling Language (AWML) in Chapter 6 describes an XML format used to exchange objects of the Augmented World. Objects can be retrieved through the Augmented World Query Language (AWQL) discussed in Chapter 7. The chapter on "Nexus Locators" (Chapter 8) explains the purpose of the Locator mechanism and gives an insight on how Nexus identifies objects. In Chapter 10 the Change Report Language is presented, which encodes the outcome of insert, update and delete operations.

Chapter 10 briefly discusses the Augmented Area Description Language. Chapter 11 on "Map Predicate Language (MapPL)" discusses how maps are queried.

We then introduce the Navigation Parameter Language (NPL) for specifying navigational query parameters (Chapter 12) and the Navigation Result Language (NRL) for results of queries formulated in NPL (Chapter 13). In Chapter 14 to Chapter 16 we describe the languages relevant for the Nexus Event Service: the Event Information (EventInfo), the Event Registration Language (ERL) and the Event Notification Language (ENL). Finally, we give a short overview of the Geocast Communication Service (Chapter 17).

2 Platform Design

The goal of the Nexus Platform is to support all kinds of context-aware applications by providing a shared global context model (the so called Augmented World Model). To achieve this goal, the platform federates local context models.

The local models contain different types of context information: representations of real world objects (e.g. streets, rooms, or persons) and virtual objects (e.g. links to digital information spaces, relations or documents). Sensors can keep the data of local context models up to date (e.g. the position of a person) (Figure 1).

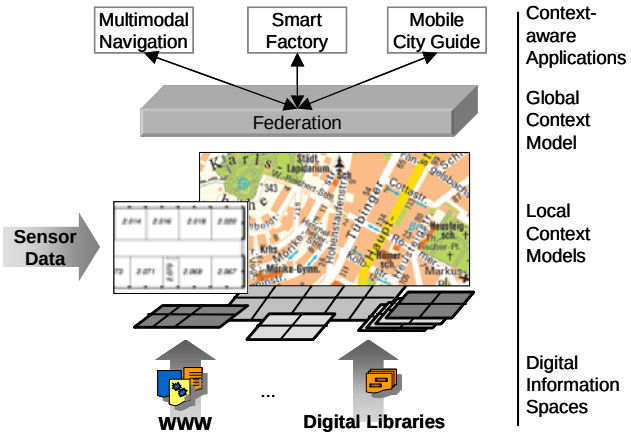

Figure 1: Vision of the Nexus platform

2.1 Overview of the Nexus Platform

The Nexus Platform consists of three tiers: Application tier, Federation tier and Service tier (Figure 2). In this chapter, we first give an overview of the different components of the platform. We then describe the three different usages an application can make of the platform – context queries, context events and value-added services – and which interfaces exist for that applications.

Though we support various types of context, spatial context is our main focus in this document.

.

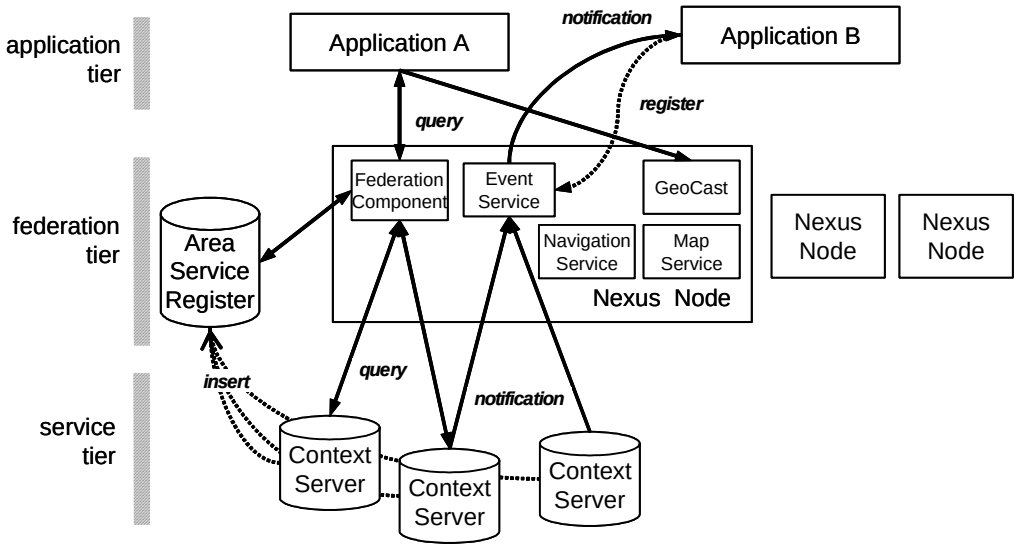

Figure 2: Architecture of the Nexus platform

2.1.1 Components of the Nexus Platform

Application Tier

A ***Nexus application*** is software that uses the Nexus platform and is not used by other applications (then we would call it a ***service***). They can be mobile (running on a portable device) or infrastructure-based (e.g. deployed to a smart environment).

A Nexus application can use the Nexus platform in three different ways. First, it can send context queries to the federation to get information about its surroundings including infrastructure, points of interest, mobile objects, and so on. Secondly, the application can register events with the event service to receive a notification when certain events or situations occur, e.g. when the user has entered a building or the temperature in a room exceeds a certain threshold. Thirdly, it can use value-added services like the Navigation Service to let common functions be processed on the Nexus Node. Some services require client-side components that facilitate the application to make use of the service.

Since the interfaces of the platform are standardized, an application can also contact every service directly by itself, skipping the Nexus node (see below) in the Federation tier. This can be reasonable for performance reasons or for specialized applications. However, in this document, we will assume that applications use the Nexus node to communicate with the platform components.

Federation Tier

A ***Nexus node*** is a server that provides the functions of the Nexus platform to applications and services. It hosts the federation component, the registration for the Event Service and can host one or many value-added services. A Nexus node does not have a persistent memory (though it will do caching).

The ***federation component*** mediates between applications and context servers. It has the same interface as a context server, but does not store data models (except for caching). Instead, it analyzes an application query, determines the context servers that could answer the query, and distributes the query to these servers. Then it combines the incoming result sets to a consistent view and returns it to the application.

Applications can register events to be observed with the ***Event Service*** that initiates the observation of the event.

Value-added services (see Section 2.2.4 on page 13) provide advanced or common functions to applications. They normally work on the federated world model and can get efficient and integrated access to the components on the Nexus node. Beside the services of the Nexus platform they can use external services, data sources or distributed components to fulfill their tasks.

For query distribution and service discovery, a Nexus node uses the ***Area Service Register (ASR)***. This service is a directory of the available local context models (called Augmented Areas or AAs) and stores the address of their context server, their object types, and a spatial region containing all their objects.

For the registration of a local context model at the ASR, the ID of the AA (Nexus Augmented Area Locator or NAAL) has to be given together with the specification of its relevant properties. The same is true for update mechanisms, whereas for deletion only the NAAL is necessary. Queries to the ASR are formulated using the AADL. They return either only NAALs of the servers that have to be addressed or additionally specifications of the Augmented Areas these servers store (if such specifications exist). Returned server descriptions are encapsulated in an AAList.

Service Tier

The Nexus platform federates local context models. The local models contain different types of context information: representations of real world objects (e.g. streets, rooms, or persons) and virtual objects (e.g. links to digital information spaces, relations or documents). Sensors can keep the data of local context models up to date (e.g. the position of a person) (Figure 1).

A ***context server*** stores such a local context model. It is comparable to a web server in the WWW. The server has to fulfill two requirements in order to be part of the Nexus Platform: it has to implement a certain interface (AWQL/AWML, allowing simple spatial queries, and the event interface) and it should be registered with its service area and object types to the Area Service Register.

If a context server supports mobile objects and an object leaves the service area, the server must be able to perform a hand-over to another suitable context server (if available). This may require a special interface (which is not fully specified yet).

There can be many different implementations of a context server. E.g, for providing large scale geographical models, we use a spatially enhanced database (Spatial Model Server). We cope with the high update rates of the positions of mobile users using a distributed main memory system (Location Service). From the Aware Home (Georgia Tech) we adapted a lightweight spatial server as context server. Even small-scale sensor platforms like the ContextCube can be used as context servers [Bauer et al 2003].

2.1.2 Context queries

With context queries, a Nexus application or service can select, insert or manipulate context models. It can select certain objects of a model and specify certain parts of the objects.

Context queries are expressed in AWQL (Augmented World Query Language), which is described in more detail in Chapter 6. The resulting objects of context queries are serialized in AWML (Augmented World Modeling Language) in Chapter 7. Results of the manipulation of context models are expressed in CRL (Change Report Language) (see Chapter 9).

2.1.3 Context events

Context events allow applications or services (event clients) to define events, e.g. situations of interest, and to be notified whenever such an event occurs (e.g. "Inform me when I come close to the University Bookstore"). If a context event is registered by an event client, the Nexus ***Event Service*** observes the event based on information from the event sources (generally context servers). If the event occurs, the event client will be notified. The interfaces and components of the Event Service are described in more detail in [Bauer 2004].

Context events are specified in “Event Notification Language (ENL)” (Chapter 15). Information about a registered event is provided by the “Event Information (EventInfo)” (Chapter 14). Context notifications are expressed in “Event Notification Language (ENL)” (Chapter 16).

2.2 Component Interfaces

2.2.1 Event Service

The purpose of the Nexus Event Service is to allow the user or application to define events, e.g. situations of interest, and to be notified whenever such an event occurs. Examples of typical events are the following:

- Inform me when I come close to the University Bookstore (e.g. because I have to pick up a book there).
- Inform me when I pass a (=any) shoe shop (e.g. because I have to buy new shoes for dancing).
- Inform me when one of my friend is close by (e.g. so that we can do something together).

Like the given examples, most of the events that are of particular interest in the context of Nexus are spatial events, i.e. events that occur when a certain spatial constellation of objects is reached. However, other events are also possible, e.g. inform me when the temperature in room 2.469 is greater than 30°C.

Event communication realizes push communication, i.e. the communication is initiated by the system and the information is pushed to the user, as opposed to query-based communication, which is pull-based, i.e. the communication is initiated by the user or application and new information only becomes available on request.

The component where the information originates is called Event Source, the component that receives the event notification is called the Event Client. The Nexus Event Service is responsible for both the observation of those events that are based on information from multiple Event Sources and the delivery of event notifications.

Event Service Architecture

Figure 3 shows the conceptual event service architecture.

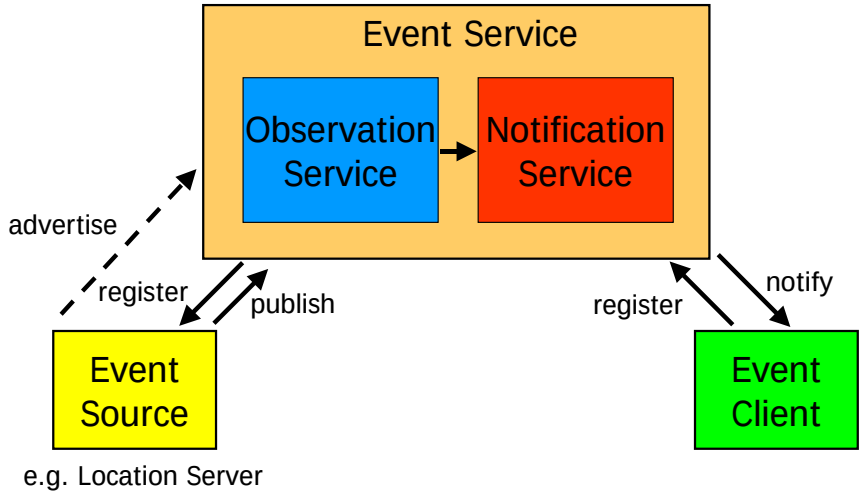

Figure 3: Conceptual Event Service Architecture

The Nexus Event Service requires that the observation of events has to be explicitly initiated by the user or application. The number of potential events is too large to be observed automatically by the Event Service. Therefore, there are two phases: the registration phase in which the observation of the event is initiated and the observation phase in which the event is being observed and the user is notified when the event has occurred. An Event Client uses the Event Registration Language (ERL), which is described in Chapter 15, to register an event. As the result of a successful registration an EventInfo, which is described in Chapter 14, is returned. The event notification, which informs the user about the occurrence of an event, is specified based on the Event Notification Language, which is described in Chapter 16

The observation of events is based on Event Sources. In Nexus the Event Sources are the Context Servers that store the model information. Currently only Location Servers are Event Sources that can observe events locally. In the future, Spatial Model Servers will also become event sources.

The Event Service itself consists of two parts, the Observation Service and the Notification Service. The Observation Service observes events for which the underlying data is distributed over multiple Event Sources so that the event can not be observed locally. The Notification Service is responsible for delivering event notifications about the occurrence of an event to all the clients interested in the event.

Event Clients register their interest in an event with the Event Service. If the event is not already being observed the observation has to be initiated with the Observation Service and/or Event Sources. When Event Sources or the Observation Service start observing an event, they advertise this to the Event Service, so that the Notification Service can set up the distribution of event notifications. When an event occurs the Event Source or the Observation Service publishes an event notification that is distributed by the Notification Service, which notifies all clients interested in the event.

Figure 4 shows how the Event Service fits into the tiers of the Nexus architecture.

The Observation Service consists of a number of Observation Nodes and Observer Placement Components. The Observer Placement Components will be responsible for optimally placing the event observation on suitable Observation Nodes and Event Sources by registering appropriate events there using ERL. For the optimal placement of the observation, the Observer Placement Component has to know, which servers store the model information. In order get this information, it queries the Area Service Register

(ASR) and possible other registers. Currently only a rudimentary static implementation of the Observer Placement Component exists. The observation of events is realized by the Observation Nodes and Event Sources. In addition to Event Sources, there may be Information Sources that Observation Nodes query when observing certain events, e.g. – for our second example in the introduction about the Event Service – what kind of shoe shops are in the vicinity of the user so that he can be notified if he comes close to one of them.

The Notification Service consists of a number of Notification Nodes that communicate with each other on a peer-to-peer basis. Event Sources hand over event notifications to the closest Notification Node, which then delivers it to all Notification Nodes with interested client applications. These Notification Nodes notify the Event Clients (applications), passing on the event notification.

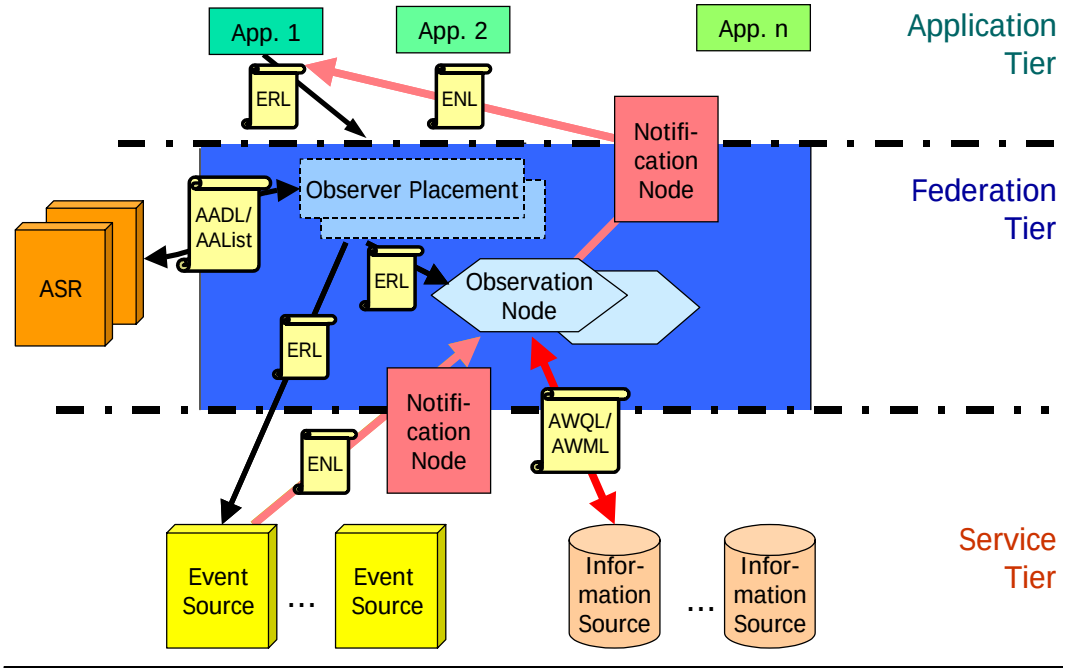

Figure 4: The Event Service in the Nexus Architecture

A more detailed description of the Event Service and the observation of events on data from distributed Event Sources can be found in [Bauer 2004].

Event Client to Event Service Interface

The Event Service provides the following interface to Event Clients:

- register
  - Parameter: ERL
  - Return Value: EventInfo
- deregister
  - Parameter: EventInfo
  - Return Value: Boolean
- refresh
  - Parameter: EventInfo
  - Return Value: Boolean

Event Service to Event Client Interface

The Event Client has to implement the following interface for the Event Service:

- notify
  - Parameter: ENL
  - Return Value: Boolean

Event Source to Event Service Interface

The Event Service provides the following interface to Event Sources:

- advertise
  - Parameter: EventInfo
  - Return Value: Boolean
- unadvertise
  - Parameter: EventInfo
  - Return Value: Boolean
- publish
  - Parameter: ENL
  - Return Value: Boolean

Event Service to Event Source Interface

An Event Source has to provide the following interface to the Event Service:

- register
  - Parameter: ERL
  - Return Value: Boolean
- deregister
  - Parameter: EventInfo
  - Return Value: Boolean
- refresh
  - Parameter: EventInfo
  - Return Value: Boolean

2.2.2 Interfaces of the ASR

The ASR provides the following interfaces.

- insert
  - Parameters: AADL-document
  - Return value: CRL-document
- update
  - Parameters: AADL-document
  - Return value: CRL-document
- delete
  - Parameter: AADL-document (only with NAAL)
  - Return value: CRL-document
- query
  - Parameter: AADL-document
  - Return value: AAList-document

2.2.3 Context Server Interfaces

Both the federation component and the context servers provide the following interface for context queries:

- insert
  - Parameter: AWML-document

  - Return value: CRL-document
- update
  - Parameter: AWQL-document
  - Return value: CRL-document
- delete
  - Parameter: AWQL-document
  - Return value: CRL-document
- query
  - Parameter: AWQL-document
  - Return value: AWML-document

2.2.4 Value-added services

In addition to the query features, a Nexus node can support value-added services that have their own interfaces and use the federated context model.

Up to now, we defined interfaces for the following value-added services:

Map Service

A map service renders maps from context models. MapPL (Map Predicate Language) which can be used to specify which objects should be included in the map and how the output should look like (size, graphics format and so on). It has a query interface:

- query
  - Parameter: MapPL-document
  - Return value: XML document containing map and description parameters

Navigation Service

The Navigation Service computes multi-modal navigation routes across the borders of local context models. NPL (Navigation Parameter Language) allows to specify the parameters like start and end position of the route. The resulting route is given in NRL (Navigation Result Language). The Navigation Service has a query interface:

- query
  - Parameter: NPL-document
  - Return value: NRL-document

GeoCast

The geocast components implement a communication service that can be used to send messages to hosts located in a certain geographic area called the target area of the message.

Our geocast system consists of three kinds of components: GeoClients, GeoNodes, and a GeoRegister.

A ***GeoClient*** is a daemon running on the client device. It is used by local applications to send and receive geocast messages.

A ***GeoNode*** is responsible for distributing geocast messages in the geographic area covered by certain sub-networks. This area is called the service area of the GeoNode. It is determined by the geographic positions of the GeoClients within these sub-networks. To distribute a geocast message, a GeoNode multicasts received messages to a certain multicast group that the GeoClients in the sub-network have joined.

The ***GeoRegister*** stores mappings of GeoNode service areas to GeoNode network addresses for all GeoNodes. To send a geocast message, the GeoNode responsible for the sending GeoClient queries the GeoRegister for all GeoNodes whose service areas overlap with the target area of the message. Then it sends the geocast message to these GeoNodes using multiple unicast messages. These GeoNodes distribute the message in

the target area as described above. At the moment, the GeoRegister is implemented as a centralized component. As soon as the AWM can handle network coverages, the GeoNodes can be modeled as objects in the AWM and the target GeoNodes can be determined by querying the Nexus Federation.

2.3 Data Modeling

2.3.1 Concept

We need a global schema (also the type definition) for enabling the federation of local context models into a global context model. Our current global schema is the result of analyzing spatial applications and application scenarios, and of projects developing applications within the Nexus project.

Our data schema is object-based, i.e. we define object types and inheritance relations between object types. The inheritance relations model is-a relationships. In our data schema, multiple inheritance is permitted. Also, Nexus objects can have multiple types (to represent the merge of multiple representations of objects).

2.3.2 Implementation

For data exchange in the Nexus platform we use XML technology. We define an XML language for the standard data exchange between components and applications (so called AWML (Augmented World Modeling Language)). To represent the data schema, we use XML Schema [XML Schema] which fulfills most of our requirements. However, because XML Schema does not allow multiple inheritance and multiple types we separate the class schema including inheritance information (NSCS) from the serialization format definition of Nexus objects (AWML). Thus the NSCS serves as a semantic definition and represents the logical organization of objects which has to be enforced as a rule set, while the AWML acts as the physical layer and as a syntactical correct instance of the semantic rules defined in the NSCS. We also separate the definition of basic types (NSAT) from the definition of complex types and attributes (NSAS). As predicates of our query language can be executed only on basic types, we are able to automatically generate the necessary code. Inclusion of the complex types into the query mechanism would have raised the complexity of the code generation process and would have also forced us to implement manually methods for each complex type used in the Nexus platform. These problems are avoided by constructing the complex types and attributes from the simple types available, thus keeping code generation efforts and complexity to a minimum.

NSAT, NSAS and NSCS represent rule sets, they are never instantiated and only used for ensuring the correct format of AWML instances.

Overall, the following four XML Schema documents define the Nexus global data schema:

- In the **Nexus Standard Attribute Types** document (NSAT) we define all basic types, executable for our query language predicates (strings, geographical data types, and so on). For geographical and temporal data types we import GML type definitions (for more information about GML see [GML]).
- In the **Nexus Standard Attribute Schema** document (NSAS) we combine basic types from NSAT to complex types and use these basic and complex types to define all simple and complex attributes (e.g. position or address) of our object types.
- In the **Nexus Standard Class Schema** document (NSCS) we define all Nexus object types and their inheritance relations. Also we define which attributes belong to which object types.

- In the **Augmented World Modeling Language** document (AWML) we define the serialization of nexus objects for data exchange.

The XML Schema documents and their import relationships are shown in Figure 5.

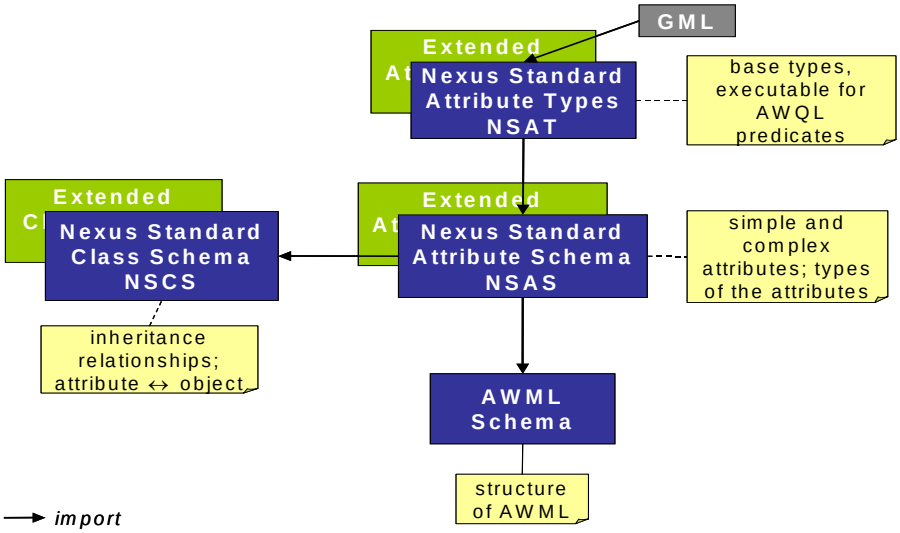

Figure 5: XML Schemas for Data Modelling and Data Exchange

Every data provider can define extensions to the global data schema. So called **Extended Class Schemas** define new object types. New object types are always refinements of the NSCS in terms of their inheritance relation and are linked to the NSCS using inheritance relationships. If some application cannot deal with a certain Extended Class Schema, the objects can be upcasted into the parent object types of the NSCS.

In addition to new object types, Extended Attribute Schemas, defining new complex types and/or attributes, and Extended Attribute Types, defining new basic types, can be given.

We do not support Extended Attribute Types so far, because this would require to extend all query processing code to cope with the new data types (serialization, comparison, indexing,...).

2.3.3 XML Examples

The following examples show characteristic clippings of our XML schemas and documents. Working examples for testing can be found in [XML Schema Definitions].

Namespace Definitions

Namespace Definitions for all following examples in this document:

```
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:gml="http://www.opengis.net/gml"
xmlns:nsat="http://www.nexus.uni-stuttgart.de/1.0/NSAT"
xmlns:nsas="http://www.nexus.uni-stuttgart.de/1.0/NSAS"
xmlns:nscs="http://www.nexus.uni-stuttgart.de/1.0/NSCS"
xmlns:awml="http://www.nexus.uni-stuttgart.de/2.0/AWML"
xmlns:awql="http://www.nexus.uni-stuttgart.de/2.0/AWQL"
xmlns:mpl="http://www.nexus.uni-stuttgart.de/1.0/MapPL"
xmlns:nol="http://nexus.informatik.uni-stuttgart.de/NOL"
xmlns:eventinfo="http://www.nexus.uni-stuttgart.de/1.0/
EventInfo"
```

```
xmlns:notify="http://nexus.informatik.uni-stuttgart.de/2.0/
EventNotificationLanguage"
xmlns:register="http://nexus.informatik.uni-stuttgart.de/2.0/
EventRegistrationLanguage"
```

3 Nexus Standard Attribute Types (NSAT)

3.1 Idea

The Nexus Standard Attribute Types schema (NSAT) defines basic types for the Nexus Augmented World. AWQL predicates can be executed only on these basic types. We define Nexus Standard Attribute Types for

- strings
- numerical data types
- geographical data types
- temporal data types

Up to now we have not defined Standard Attribute Types for

- XML documents including access with XPath
- list types (currently coded as strings)
- set types

Extended Class Schemas are not allowed to define an Extended Attribute Types schema so far.

3.2 Examples

The following examples are cuttings of the XML schema document (see [XML Schema Definitions]). First the Nexus Standard Attribute Type for string values:

```
<simpleType name="NexusStringType">
  <restriction base="string"/>
</simpleType>
```

The Nexus Standard Attribute Type for geometry values:

```
<complexType name="NexusSrscodeType">
  <restriction base="nsat:nonNegativeInteger"/>
</complexType>
<complexType name="NexusWktType">
  <simpleContent>
    <extension base="string">
      <attribute name="srscode" type="nsat:NexusSrscodeType"/>
    </extension>
  </simpleContent>
</complexType>
<complexType name="NexusGeometryType">
  <choice>
    <element name="WKT" type="nsat:NexusWktType"/>
    <element ref="gml:_Geometry"/>
  </choice>
</complexType>
```

Please note that this example is not an instance but a listing of the type definition. Instances of NSAT are never used, because NSAT is only used for type checking.

4 Nexus Standard Attribute Schema (NSAS)

4.1 Idea

The Nexus Standard Attribute Schema (NSAS) defines the Nexus attributes which are put together to form Nexus objects as defined in the Nexus Standard Class Schema. The Nexus Standard Attribute Schema is imported in the NSCS (Chapter 5) and AWML (Chapter 6).

In the Nexus Standard Attribute Schema, base data types from the Nexus Standard Attribute Types document are put together to complex attributes. Every attribute contains a value and a meta tag.

```
<anyNexusAttribute>
  <value>...</value>
  <meta>...</meta>
</anyNexusAttribute>
```

The value tag contains the actual Nexus attribute value (simple or complex). The meta tag contains metadata attributes. Metadata attributes are defined in the Nexus Standard Attribute Schema just like Nexus attributes.

The depth of complex attributes (i.e. complex values or complex metadata attributes) is basically not limited, but the nesting has to be defined within the attribute or metadata attribute definition, respectively. Every leaf type has to be one of the Nexus Standard Attribute Types.

4.1.1 Metadata

Metadata is data on data. In this section we look at metadata on Nexus objects and on Nexus attributes; metadata on Augmented Areas is discussed in Chapter 10.

Metadata on Nexus objects and attributes has the following characteristics:

- There are simple and complex meta data attributes.
- When computing the result set of an AWQL query, restrictions on (specific) metadata attributes should be considered.
- The return set of an AWQL query should contain (specific) metadata attributes.

As mentioned above, metadata attributes are defined in the Nexus Standard Attribute Schema just like Nexus attributes. These metadata attributes can be used as metadata attributes on Nexus objects as well as on Nexus attributes.

4.2 Examples

Example of a Nexus attribute with a simple value:

```
<complexType name="NexusGeometryAttributeType">
  <sequence>
    <element name="value" type="nsat:NexusGeometryType"/>
    <element ref="nsas:meta" minOccurs="0"/>
  </sequence>
</complexType>
```

```
<element name="extent"
   type="nsas:NexusGeometryAttributeType"
   substitutionGroup="nsas:NexusAttribute"/>
```

Example of a Nexus attribute with a complex value:

```
<complexType name="NexusAddressAttributeType">
   <sequence>
      <element name="value">
         <complexType>
            <sequence>
               <element name="street" type="nsat:NexusStringType"
                  minOccurs="0"/>
               <element name="number" type="nsat:NexusStringType"
                  minOccurs="0"/>
               <element name="city" type="nsat:NexusStringType"
                  minOccurs="0"/>
               <element name="zipCode" type="nsat:NexusIntegerType"
                  minOccurs="0"/>
            </sequence>
         </complexType>
      </element>
      <element ref="nsas:meta" minOccurs="0"/>
   </sequence>
</complexType>

<element name="address" type="nsas:NexusAddressAttributeType"
   substitutionGroup="nsas:NexusAttribute"/>
```

Example of a simple metadata attribute:

```
<element name="validTime" type="nsat:NexusTimeType"
   substitutionGroup="nsas:NexusMetaAttribute"/>
```

Please note that this example is not an instance but a listing of the complex type definition. Instances of NSAS are never used.

5 Nexus Standard Class Schema (NSCS)

5.1 Idea

We need a global data schema for enabling the federation of local context models into a global context model. Our current Nexus Standard Class Schema (see [XML Schema Definitions]) is the result of analyzing and developing spatial applications and application scenarios.

The current NSCS includes object types for real word objects (like streets, buildings, mobile objects) as well as object types for virtual objects that link to digital information spaces (e.g. virtual information towers). For high flexibility, most attributes in the NSCS are optional.

Extensions to the NSCS can be defined by every data provider. So called Extended Class Schemas are linked to the NSCS using inheritance relationships. If some application cannot deal with a certain Extended Class Schema, the objects can be upcasted into the parent object types of the NSCS. In addition to new object types, Extended Attribute Schemas and Extended Attribute Types can be part of an Extended Class Schema.

In the NSCS and in Extended Class Schemas, multiple inheritance is permitted. Because of multiple inheritance and the possibility of merging multiple representations, Nexus objects can have multiple types.

Because it is impossible to define multiple inheritance in XML Schemas, we specify the inheritance information in **appinfo** (application information) tags.

A human readable version of the NSCS can be found under [AWS].

5.2 Example

```
<complexType name="NexusDataObject">
  <annotation>
    <documentation>object with a NOL</documentation>
    <appinfo>
      <extension base="nscs:NexusObject"/>
    </appinfo>
  </annotation>
  <complexContent>
    <extension base="nscs:NexusObject">
      <sequence>
        <element ref="nsas:nol"/>
        <element ref="nsas:kind"/>
        <element ref="nsas:name" minOccurs="0"/>
        <element ref="nsas:description" minOccurs="0"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

Please note that this example is not an instance but a partial listing of the schema definition. Instances of NSCS are never used.

6 Augmented World Modeling Language (AWML)

6.1 Idea

The Augmented World Modeling Language (AWML) is used to represent information about the Augmented World (AW). An AWML document consists of a sequence of data objects of the AW.

The Nexus Standard Class Schema is the underlying class model of the AW. It defines the object types and inheritance relationships between object types as well as their attributes.

In AWML, complex attributes (i.e. complex attribute values or complex metadata attributes) are allowed. The depth is basically not limited, but the nesting has to be defined within the attribute or metadata attribute definition in the Nexus Standard Attribute Schema, respectively. Every leaf type has to be one of the Nexus Standard Attribute Types (Figure 6).

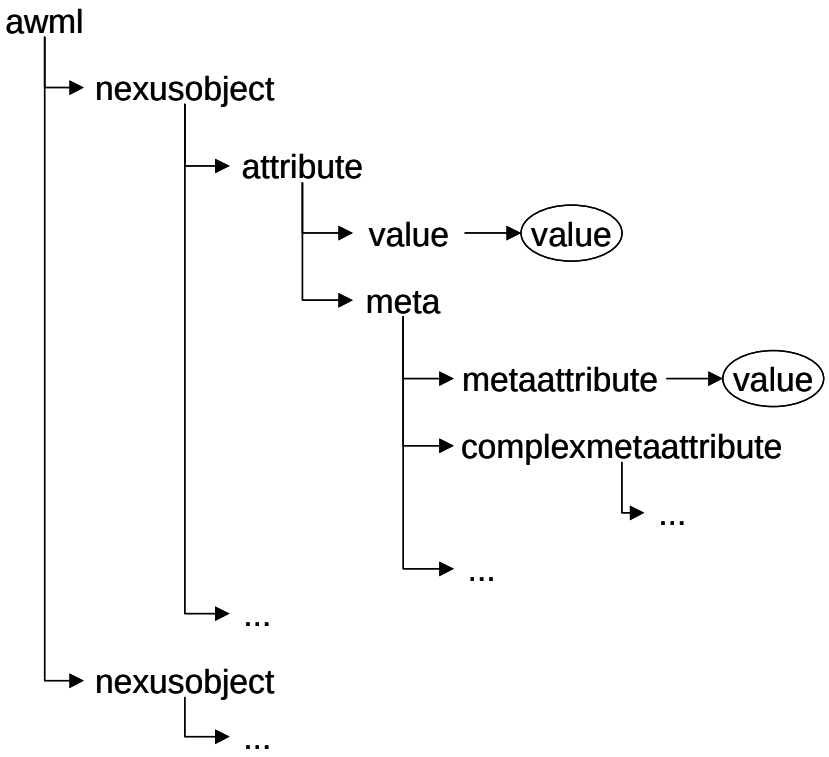

Figure 6: Structure of an AWML document

Leafs or parts of an AWML document tree are specified giving the path. The following example denotes all leaf nodes of a simple metadata attribute included in a specific attribute.

```
someattribute.meta.somesimplemetaattribute
```

This notation for leafs and parts of AWML document trees is used in the restriction and include tags in AWQL.

The Federation tries to detect and merge multiple representations of Nexus objects. Merged objects can have multiple values of one attribute.

Also multiple values of an attribute of one object on a Context Server can exist, e.g. different values for position at different times (meta data).

6.2 Example

This example contains three Nexus objects with different attributes. For some objects and attributes metadata attribute values are given.

```
<awml:awml>
  <awml:nexusobject>
    <nsas:type><nsas:value> Building </nsas:value></nsas:type>
    <nsas:NOL><nsas:value>...some NOL..</nsas:value></nsas:NOL>
    <nsas:address>
      <nsas:value>
        <nsas:street> Universitätsstraße </nsas:street>
        <nsas:city> Stuttgart </nsas:city>
      </nsas:value>
    </nsas:address>
    <nsas:extent>
      <nsas:value>
        <nsas:gml>
          ...GML representation of the extent...
        </nsas:gml>
      </nsas:value>
    </nsas:extent>
  </awml:nexusobject>
  <awml:nexusobject>
    <nsas:type>
      <nsas:value> MobileObject </nsas:value>
    </nsas:type>
    <nsas:name>
      <nsas:value> Daniela Nicklas </nsas:value>
    </nsas:name>
    <nsas:pos>
    <nsas:value>
        <nsas:gml>...some position value...</nsas:gml>
      </nsas:value>
      <nsas:meta>
        <nsas:validTime>
          ...some timestamp value...
        </nsas:validTime>
      </nsas:meta>
    </nsas:pos>
    <nsas:pos>
    <nsas:value>
        <nsas:gml>...some other position value...</nsas:gml>
      </nsas:value>
      <nsas:meta>
        <nsas:validTime>
          ...some other timestamp value...
        </nsas:validTime>
      </nsas:meta>
    </nsas:pos>
  </awml:nexusobject>
```

```
  <awml:nexusobject>
    <nsas:type>
      <nsas:value> VirtualInformationTower </nsas:value>
    </nsas:type>
    <nsas:meta>
      <nsas:validTime>
        ...only during the fair...
      </nsas:validTime>
    </nsas:meta>
  </awml:nexusobject>
</awml:awml>
```

Please note that this example is different from those used in NSAT, NSAT and NSCS because the example given above is a listing of an actual instance.

7 Augmented World Query Language (AWQL)

7.1 Idea

Queries for Context Servers and the Federation are formulated in a query-language (AWQL), to select certain objects of a model and specify certain parts of the objects. The result of a query is a subset of the Augmented World (a set of objects) that match the conditions described by the query and that are transformed as stated in the query.

AWQL is also used to select the objects affected by updates or deletions.

The Federation may support a set of higher-level functions in addition to AWQL, e.g. for generating maps (value-added services).

An AWQL-query may contain the following parts.

- **aa**: A Nexus Augmented Area Locator (NAAL, see chapter 8 “Nexus Locators”) specifying the specific target of the query.
- **ecs**: A list of Extended Class Schema (ECS) identifiers. The union of these ECSs plus the Standard Class Schema (SCS) defines the "Schema" used by and understood by the querying instance (the application or the federation).
- **srs**: A list of spatial reference systems acceptable to the sender.
- **geoDataFormat**: A list of geographic data formats acceptable to the sender.
- **restriction**: The restriction is a boolean predicate that the objects of the result set must meet.
- **nearestNeighbor**: The application can restrict the result of its query to contain only the n objects closest to a given position.
- A filter element defines which operation has to be executed on the objects of the result set. These elements are mutually exclusive.
  - **include** or **exclude**: Restricts the objects of the result set to a return set. This element can be used to delete attributes of the returned objects. The return value is an AWML document.
  - **update**: This element determines the attributes to update and their new values. The return value is a CRL document.
  - **append**: This element determines that a new attribute instance should be appended to existing attribute instances. The return value is a CRL document.
  - **delete**: This element determines the deletion of the result set objects. The return value is a CRL document.
- Ordering restrictions for the return set (not implemented yet)

7.1.1 Defining Augmented Areas (aas)

The list of **<aa>** elements specifies the Augmented Areas (AAs) that are the specific target of the query. An application can use this to directly query a specific data source omitting the "context server discovery" part of the federation layer. Such a data source can be known in advance to the application e.g. because it stores the application’s private data.

When querying that context server the federation layer can use this feature to address only a specific AA of all AAs stored on that context server, which might help the context server to optimize the query processing.

7.1.2 Defining Extended Class Schemas (ecs)

The **ecs** tag contains an Extended Class Schema (ECS) identifier. The union of these ECSs plus the Nexus Standard Class Schema (NSCS) defines the "Schema" used by and understood by the querying instance (the application or the federation).

The idea behind this is that the federation gets to know what the application is interested in and can thus do some complex federation activities (integrating data from other data sources that might have a syntactically different but semantically similar ECS). On the context server side all ECS objects that are within the "scope" of the application can be returned "as is", while objects outside the applications scope have to be upcasted (transformed) to the closest SCS object type.

7.1.3 Restriction

The restriction is a boolean predicate that the objects of the result set must meet. Its elements are comparisons between attributes and values, furthermore geometric relations (within, intersects) and temporal relations (before, after, overlaps) between attributes and values.

Attributes and values for every predicate are given in the following manner:

```
<predicate>
   <target> path to a leaf in an AWML document tree </target>
   <referenceValue> comparison value </referenceValue>
</predicate>
```

In this context, attribute means the target attributes for metadata attritbutes of an object as well as metadata attribute values of a specific attribute or an object. Attribute values and metadata attribute values have to be given by their path in an AWML document tree (see Chapter 6 for more information). Reference values of predicates have to be values of the basic types defined in the Nexus Standard Attribute Types (NSAT) (Chapter 3).

If two or more predicates observe the same Nexus attribute, then they have to observe the same instance of this attribute. For example there is an AND-predicate combining two other predicates observing the same Nexus attribute. To make the AND conjunction true there needs to be at least one attribute instance where both of the predicates are true at the same time.

Restriction Operators

All predicates within the restriction have boolean return values.

Boolean conjunctions with standard semantics:

- and
- or
- not

Comparison predicates on standard data types (please note that if an object has multiple instances and thus multiple values then the comparison predicate becomes true if at least one of those instances or values satisfies the comparison predicate):

- equal
  - exact match on strings, numerical data types, booleans
  - semantics for type values: equal (type A, type B) iff (A = B) or (A is a subclass of B)
    Notice: returned objects will not be upcast, they remain their subclass.
- like
  - substring matches on strings
- greater
  - one value (target) is greater than another value (referenceValue)

  - only on numerical data types
- less
  - target is less than referenceValue
  - only on numerical data types

Comparison operators on geographic data types:

- within (formerly: inside)
  - one geometry (target) is completely contained in another geometry (referenceValue) (there is no point of the target geometry outside of the referenceValue geometry)
- intersects (formerly: overlaps)
  - target intersects with referenceValue

Comparison operators on temporal data types:

- temporalBefore
  - less than on temporal data types
  - one temporal value (target) is before another temporal value (referenceValue), that means, target.end is before referenceValue.begin
- temporalBeforeBegin
  - same as temporalBefore, but compares target.begin and referenceValue.begin
- temporalAfter
  - greater than on temporal data types
  - target.begin is after referenceValue.end
- temporalAfterEnd
  - same as temporalAfter, but compares the end time point value of target
- temporalEqual
  - exact match
- temporalEqualBegin
  - same as temporalEqual, but compares the start time point value of target
- temporalEqualAfter
  - same as temporalEqual, but compares the end time point value of target
- temporalIntersects
  - analog to intersects
  - at least one time point of the target temporal value has to be within the referenceValue temporal value

7.1.4 Filter

A filter element defines which operation has to be executed on the objects of the result set. An AWQL query may contain at most one of the following filter elements.

include or exclude

The **include** and **exlude** tags restrict the attributes of a result set to a return set. With exclude you can specify which attributes or part of attributes of the result set objects should not be in the return set. All other attributes are included.

With include you can specify which attributes or part of attributes of the result set objects should be in the return set. All other attributes are excluded. Within include other restrictions on the result set attributes can be given, e.g. for choosing specific attribute instances. Within the include-restriction the same predicates as in the normal restriction can be used. A restriction within an exclude element makes no sense as the exclude element excludes all instances of the given attribute. To exclude only some instances of an attribute (by using predicates) the include element has to be used.

The return value of the query is an AWML document.

update

This element determines the attributes to update. New values are given inside the update tag. An update removes all previous attribute instances. The return value is a CRL document.

append

This element determines new attribute instances to append to existing attribute instances. New instances are given inside the append tag. The return value is a CRL document. This results in multiple instances and also multi-values of the object.

delete

This element determines the deletion of the result set objects. The return value is a CRL document.

7.2 Examples

This is the solution for the famous "Italian-restaurant-problem": The result contains the name, the menu and the position of the closest five Italian restaurants. The example supposes that the restaurant object has at least three attributes (**name**, **menu**, and **pos**, the position).

```
<awql:awql xmlns:sr="nexus://nexusschemas.org/SpecialtyRestau-
rants">
  <awql:aa>nexus://guide.stuttgart.de/gourmet/...</awql:aa>
  <awql:aa>nexus://michelin.com/europe/...</awql:aa>
  <awql:ecs> nexus://nexusschemas.org/SpecialtyRestaurants
    </awql:ecs>
  <awql:geoDataFormat>
    WKT
  </awql:geoDataFormat>
  <awql:restriction>
    <awql:or>
      <awql:and>
        <awql:equal>
          <awql:target> nsas:type.nsas:value </awql:target>
          <awql:referenceValue> restaurant
            </awql:referenceValue>
        </awql:equal>
        <awql:equal>
          <awql:target> nsas:style.nsas:value </awql:target>
          <awql:referenceValue> italian </awql:referenceValue>
        </awql:equal>
      </awql:and>
      <awql:equal>
        <awql:target> nsas:type.nsas:value </awql:target>
        <awql:referenceValue> sr:ItalianRestaurant
          </awql:referenceValue>
      </awql:equal>
    </awql:or>
  </awql:restriction>
  <awql:nearestNeighbor num="5">
    <awql:target> nsas:pos.nsas:value </awql:target>
    <awql:referencePoint>
      <nsat:WKT>...WKT representation of my position value...
        </nsat:WKT>
    </awql:referencePoint>
  </awql:nearestNeighbor>
```

```
  <awql:include>
    <awql:target> nsas:pos.nsas:value </awql:target>
  </awql:include>
  <awql:include>
    <awql:target> nsas:menu.nsas:value </awql:target>
  </awql:include>
  <awql:include>
    <awql:target> nsas:name.nsas:value </awql:target>
  </awql:include>
</awql:awql>
```

With the next example, an application can find out what information is available on the web about the building I'm pointing at with my telefinger. This task actually requires two queries: one to get the position (or better the extent) of the building and the second to find the web pages associated with this position. The application has to use the result of the first query to compose the second query.

```
<awql:awql>
  <awql:restriction>
    <awql:intersects>
      <awql:target> nsas:extent.nsas:value </awql:target>
      <awql:referenceValue>
        <nsat:gml>...GML representation of an object describing
        the beam of my telefinger...</nsat:gml>
      </awql:referenceValue>
    </awql:intersects>
  </awql:restriction>
  <awql:nearestNeighbor num="1">
    <awql:target> nsas:extent.nsas:value </awql:target>
    <awql:referenceValue>
      <nsat:gml>...GML representation of an object
      describing my position...</nsat:gml>
    </awql:referenceValue>
  </awql:closest>
  <awql:include>
    <awql:target> nsas:extent.nsas:value </awql:target>
  </awql:include>
</awql:awql>

<awql:awql>
  <awql:restriction>
    <awql:and>
      <awql:equal>
        <awql:target> nsas:type.nsas:value </awql:target>
        <awql:referenceValue> WebPage </awql:referenceValue>
      </awql:equal>
      <awql:within>
        <awql:target> nsas:pos.nsas:value </awql:target>
        <awql:referenceValue>...result of first query...
          </awql:referenceValue>
      </awql:within>
    </awql:and>
  </awql:restriction>
  <awql:include>
    <awql:target> nsas:uri.nsas:value </awql:target>
  </awql:include>
</awql:awql>
```

An example for multiple attribute values: position traces of mobile objects. The result contains all traces of mobile objects for the time period "yesterday" and the geographical range "Stuttgart".

```
<awql:awql>
  <!-- select objects instances -->
  <awql:restriction>
    <awql:and>
      <awql:equal>
        <awql:target> nsas:type.nsas:value </awql:target>
        <awql:referenceValue> mobileObject
          </awql:referenceValue>
      </awql:equal>
      <awql:intersects>
        <awql:target> nsas:pos.nsas:value </awql:target>
        <awql:referenceValue>...Stuttgart...
          </awql:referenceValue>
      </awql:intersects>
      <awql:temporalIntersects>
        <awql:target> nsas:pos.nsas:meta.nsas:validTime
          </awql:target>
        <awql:referenceValue>...yesterday...
          </awql:referenceValue>
      </awql:temporalIntersects>
    </awql:and>
  <!-- select attribute instances -->
  </awql:restriction>
  <awql:include>
    <awql:target> nsas:pos </awql:target>
    <awql:include>
      <awql:target> nsas:value </awql:target>
      <awql:target> nsas:meta.nsas:validTime </awql:target>
    </awql:include>
    <awql:restriction>
      <awql:and>
        <awql:intersects>
          <awql:target> nsas:pos.nsas:value </awql:target>
          <awql:referenceValue>...Stuttgart...
            </awql:referenceValue>
        </awql:intersects>
        <awql:temporalIntersects>
          <awql:target> nsas:pos.nsas:meta.nsas:validTime
            </awql:target>
          <awql:referenceValue>...yesterday...
            </awql:referenceValue>
        </awql:temporalIntersects>
      </awql:and>
    </awql:restriction>
  </awql:include
</awql:awql>
```

An example for updates: free spot quantity of a parking site. This update could be triggered by an event, making any user interaction unnecessary.

```
<awql:awql>
  <awql:restriction>
    <awql:equal>
      <awql:target> nsas:NOL.nsas:value </awql:target>
      <awql:referenceValue>
        someNOL
      </awql:referenceValue>
```

```
      </awql:equal>
    <awql:restriction>
    <awql:update>
      <nsas:freeSpotQuantity>
        <nsas:value>...new value...</nsas:value>
      </nsas:freeSpotQuantity>
    </awql:update>
  </awql:awql>
```

8 Nexus Locators

8.1 Idea

The purpose of the Nexus Locator mechanism is to assign a unique ID to every object in the Augmented World described by AWML. These IDs need to satisfy the following requirements:

1. be globally unique
2. be constant over the full object lifecycle
3. create new IDs without a bottleneck or single point of failure (also when the application is in disconnected mode or has no internet access available)
4. allow easy and efficient object discovery (e.g. to efficiently query for an objects information given its ID)
5. objects should be able to change their location or position of storage
6. multi-representations of an object can be recognized from the IDs

These requirements contain (several) conflicting demands, e.g. the second and fourth combined with the fifth form a contradiction, as it inherently proposes the encoding of a server name into the ID which would need to change after the first move of the mobile object.

As a solution we chose to separate the ID into two parts: The first part contains a service locator for easier object discovery and the second part contains the ID itself. This ID is constructed from a timestamp and the MAC address of the device. Naturally this mechanism does only function correctly if there is only one instance of the ID generator on one device.

8.2 Object Hierarchy

The class hierachy of the Nexus locators classes is shown in Figure 7. It consists of ten different classes. Three classes are abstract base classes (NEL, NOL and OtherObject-Locator), two classes store NOLs for objects of the AWM (StaticObjectLocator and MobileObjectLocator), and two classes are used for administrative purposes (NAAL and SpaSeLocator). The remaining three classes (TemplateLocator, PredicateLocator and NotificationLocator) belong to the Event Service and are used to describe event types, instances of events and their occurrences.

The Nexus Entity Locator (NEL) is the base class for all locators used in Nexus. It knows about the target server that is referenced by this locator. Nexus Object Locator (NOL) is the base class for all locators referencing an object of the AWM. OtherObject-Locator is the base class for all object locators that use a fixed prefix instead of an Augmented Area ID.

The StaticObjectLocator is a NOL identifying a non-moving object. The object resides within an Augmented Area that is identified by its ID. As the object does not move, it also does not change its Augmented Area ID. Different Augmented Areas may overlap, and the same object (identified by its object ID) may be stored in more than one Augmented Area. In that case the object has multiple representations. Corresponding representations can be glued together by the matching object ID, while the server and the Augmented Area ID part of the NOL can obviously be different.

MobileObjectLocator is a further subclass of OtherObjectLocator and represents a general specialization for all unique, mobile objects. Instead of an AAID it uses the string ".home" as an ID part to specify that it uses its home server. This home server knows on which leaf location server the object currently is stored on depending on the object's geographical location. Currently there is no implementation of the home server; the management of mobile objects is a ongoing research topic and is therefore not reflected in Chapter 2.

The SpaSeLocator identifies a Context Server that stores objects of the AWM. It provides all information needed to establish connections to this server, e.g. to issue a query to it.

The Nexus Augmented Area Locator (NAAL) refers to an Augmented Area, which is a logical partition of a Context Server. Augmented Areas (AAs) are characterized by a coverage region and a set of supported AWS types, and they are identified by a unique ID called Augmented Area ID. Objects of an Augmented Area have to reside within its coverage region and their type has to be contained in the set of supported types.

The TemplateLocator class refers to a generic event type without specifying its event parameters, e.g. OnEnterArea or OnMeeting. For each possible event type a correlating instance of the TemplateLocator exists in the Nexus platform. The PredicateLocator, a subclass of the TemplateLocator class, refers to actual instances of the generic events which were registered to be observed and thus specifies its parameters, e.g. coordinates for the observed area. Finally, the NotificationLocator, as a subclass of PredicateLocator, identifies an observed event that fits the given predicate and also identifies the server which observed the event.

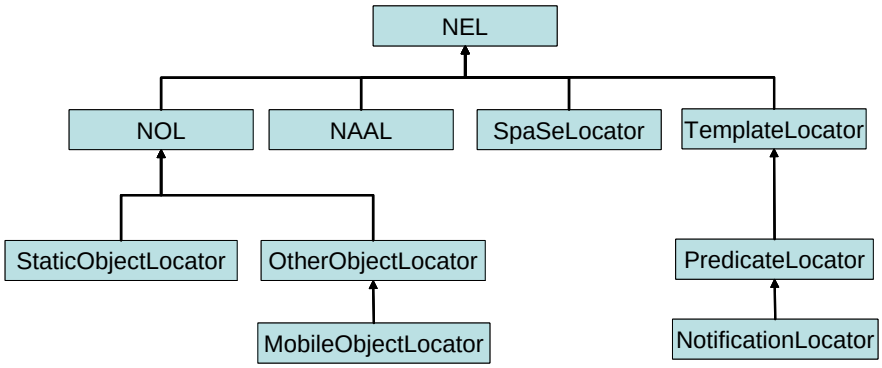

Figure 7: Object Hierarchy

8.3 Examples

8.3.1 Nexus Entity Locator (NEL)

A typical NEL has the following structure:

```
nexus:<schema_id>:<endpoint>|<service>
```

Example:

```
nexus:http://myservice.com:8080/soap/servlet/
rpcrouter|urn:QueryComponent
```

8.3.2 SpaSeLocator

The SpaSeLocator string consist only of the server endpoint.

Example:

nexus:http://democlient1:8080/soap/servlet/rpcrouter

8.3.3 Nexus Augmented Area Locator (NAAL)

The NAAL consists of a basic service part followed by an AAID (which is a 32 character long hex UUID).

Example:

nexus:http://democlient1:8080/soap/servlet/rpcrouter|urn:QueryComponent|0x734beff271e711d88a9800054e433c3a

8.3.4 Nexus Object Locator (NOL)

A NOL follows the schema for NELs given above with the addition of the AAID and object ID.

nexus:<schema_id>:<endpoint>|<service>|<aaid>/<object_id>

If the **schema_id** equals "http" then **endpoint** has the following structure:

//<servername>:<port>/<appendix>

Example:

nexus:http://myservice.com:8080/soap/servlet/rpcrouter|urn:QueryComponent|.home/0x12341234123412341234123412341234

8.3.5 TemplateLocator

The TemplateLocator has the following structure:

nexus:<schema_id>:<endpoint>|<event identifier>|<UUID>

Example:

nexus:http://eventservice.com:8080/onEnterArea|urn:OnEnterArea|0x7324a7b145c7e98f0543e6a3244885b3

8.3.6 StaticObjectLocator

A StaticObjectLocator may have multiple representations and needs to identify its instance by supplying an AAID for each representation. Only one representation per Augmented Area is possible, but Augmented Areas are not required to be disjunct. StaticObjectLocators thus do not have a fixed AAID.

Example:

nexus:http://democlient1:8080/soap/servlet/rpcrouter|urn:QueryComponent|0x734beff271e711d88a9800054e433c3a/0x4c602221c67911d79479080020a23633

8.3.7 OtherObjectLocator

OtherObjectLocator was introduced as a general abstract class serving as a base class for all future objects which require a fixed AAID. At the moment the only existing subclass is MobileObjectLocator.

8.3.8 MobileObjectLocator

All MobileObjectLocator instances have the fixed AAID ".home" instead of the typical 32 character UUID and thus may not have multi-representations.

Example:

nexus:http://democlient1:8080/soap/servlet/rpcrouter|urn:QueryComponent|.home/0x43f16facb50111d7be0c080020a23ebb

8.3.9 PredicateLocator

The PredicateLocator inherits from TemplateLocator and has an additional unique ID (32 character long UUID).

nexus:<schema_id>:<endpoint>|<event identifier>|<UUID>/<UUID>

Example:

nexus:http://observation_management.de:8080/onEnter-Area|urn:OnEnterArea|0x7324a7b145c7e98f0543e6a3244885b3/0x847295723910a8978e8907f7283ac892

8.3.10 NotificationLocator

The NotificationLocator inherits from PredicateLocator and has an additional local counter that counts the locally observed events that fit a given predicate.

nexus:<schema_id>:<endpoint>|<event identifier>|<UUID>/<UUID>/<counter>

Example:

nexus:http://eventsource123.de:8080/onEnterArea|urn:OnEnter-Area|0x7324a7b145c7e98f0543e6a3244885b3/0x847295723910a8978e8907f7283ac892/234

9 Change Report Language (CRL)

9.1 Idea

Every insert/update/delete operation on a Nexus object produces a change report as a result. The CRL result specifies success or failure of the operation on every concerned object.

9.2 Examples

A CRL document contains information about successful and failed changes at the same time.

```
<changereport>
  <success>
    <object>nexus:XXX</object>
    <object>nexus:YYY</object>
  </success>
  <failure>
    <object>nexus:ZZZ</object>
  </failure>
</changereport>
```

10 Augmented Area Description Language (AADL)

10.1 Idea

The AADL (Augmented Area Description Language) is used to insert, update, delete and query the metadata of Augmented Areas in the Area Service Register (ASR). The AADL data structure is used for both input and output of data to and from the ASR. Insert, update, and delete use a plain AADL structure and return CRLs, while query accepts an additional <resultspec> tag within the AADL structure and returns a list of AADL structures.

AADL contains different parameters for these purposes. The <naal>-tag defines the Nexus Augmented Area Locator which uniquely identifies an Augmented Area (AA) and is used for insert, update and delete procedures. The <area> and <awschema>-tags comprise the AA model specifications that are either asked for by the Federation or that are given by the Context Server that registers a new or updates an existing AA Model at the ASR. The <area> tag defines the geographic region that the AA covers, and the <awschema> tag specifies what kind of objects can be found there.

Queries return a list of <aadl> elements, one for each AA that matches the query. The <resultspec>-tag in the query defines, what information is to be returned from the ASR as a result of a query, i.e. which of the above described tags are included in each returned <aadl> element. This gives the Federation the possibility to exploit all information stored in the ASR and may render some queries to Context Servers unnecessary. A query AADL structure may contain as little as a single <naal> tag to retrieve detailed information on this AA. By default query answers contain only NAALs.

Insert, update and delete procedures return CRLs so that the success or failure of these mechanisms can be observed.

There are still some open issues to consider:

- Do we have floating or discrete level-of-detail (lod)-specifications?
- Should the scale of surveillance be attached as an attribute to the <objectclasses> tag?
- Do we have to specify the spatial extent on the level of object classes? This could be of relevance, if an object class does not cover the whole area of the Augmented Area it belongs to.

10.2 Examples

10.2.1 Query

This example defines a query to the ASR, that asks for the object classes "VIT" and "Restaurant" in the given area. As a result, not only the Naal of the appropriate Augmented Area, but also its extent should be returned as specified by the <resultspec> tag.

```
<aadl>
  <!-- by default only NAALs are returned, here additionally the
       area is returned -->
  <resultspec area = "yes"/>
  <area>
```

```
      <gml>
        <Polygon srsName ="4336">
          <outerBoundaryIs>
            <LinearRing>
              <coordinates>
              54…, 37…, etc.
              </coordinates
            </LinearRing>
          <outerBoundaryIs>
        </Polygon>
      </gml>
    </area>
    <awschema>
      <objectclass type = "VIT" lodmin = "2"lodmax = "5"/>
      <objectclass type = "Restaurant"  lodmin = "2"lodmax = "5"/
    </awschema>
</aadl>
```

Semantically all Augmented Areas qualify for the above query if their area intersects with the given area and if at the same time they contain one of the given object classes.

10.2.2 Result

The result of the query stated above is encapsulated in AAList. It returns the NAALs of the appropriate AA Models and also their extension using the <area> tag.

```
<aalist>
  <aadl>
    <naal>
    nexus://baden-wuerttemberg.de/stuttgart/restaurants
    </naal>
    <area>
      <gml>... </gml>
    </area>
  </aadl>
  <aadl>
    <naal>
    nexus://baden-wuerttemberg.de/stuttgart/stadtmitte/VIT
    </naal>
    <area>
      <gml>... </gml>
    </area>
  </aadl>
</aalist>
```

10.2.3 Insert

Here, an insert operation that registers a new AA Model at the ASR is specified:

```
<aadl>
  <naal> nexus://....</naal> <!--for insert,update, delete-->
  <area>
    <gml>....</gml>
  </area>
  <awschema>
    <objectclass type = "VIT"
      lod = "2" lodmin = "2" lodmax = "5"/>
    <objectclass type = "Restaurant"
      lod = "2" lodmin = "2" lodmax = "5"/>
  </awschema>
</aadl> <!--returns ChangeReports -->
```

11 Map Predicate Language (MapPL)

11.1 Idea

In order to generate a map, the Federation not only needs a specification of map layers and other objects to be included in the map but also some knowledge on how the output should look like. While the map and objects specification is done by an AWQL query or by providing the model data itself in AWML format, the constraints on the output are specified using the Map Predicate Language (MapPL).

Using MapPL a map can either be generated as an image map or as a vector map. For an image map the following attributes must be specified:

- The size in pixels (height and width).
- The graphics format (gif, jpeg, png, tiff or bmp).

Optionally the following attributes can be specified for the generation of image maps:

- The color depth in bits per pixel.
- A color palette: if the map is not generated in true color mode, a palette specification can be provided or a color palette will be calculated automatically.
- Greyscale image map: It is also possible to generate a greyscale image map which may be better readable on devices with limited display capabilities.

For vector maps only the format (svg or vml) can be specified at present.

Selected features on the map can be assigned to a color which is useful for highlighting them. Features on the map are selected by the specification of the respective Nexus object types.

Additionally a featuremap can be generated which correlates areas in the generated image or vector map (e.g. a star marking the position of a VIT) with the corresponding Nexus objects. Again, the selection of the features in the map is accomplished by the specification of the respective Nexus object types. This featuremap allows for the application-specific rendering of the specified Nexus objects and can e.g. be used to turn the map picture into a clickable HTML imagemap very easily.

In AWQL objects can only be selected as a whole which may lead to problems if the extent of an object is large compared to the displayed map window. Using the clipping tag this map window can be specified more precisely allowing objects at the border of the generated map to be cropped. The clipping area can be specified either in the Well-Known Text format (WKT) or by means of the Geography Markup Language (GML). Since most devices intended to display maps have rectangular display areas, it is reasonable to specify the clipping area as a rectangle. However, MapPL is not limited to the definition of rectangular clipping areas.

11.2 Examples

This example specifies that the generated map should be delivered as a 640x480 jpeg picture. Additionally a feature map describing the location and extent of VITs displayed in the map should be generated.

```
<mpl:MapPL>
  <mpl:imagemapspec>
    <mpl:imagesize width="640" height="480"/>
    <mpl:imageencoding format="jpeg" bits="24"/>
 </mpl:imagemapspec>
 <mpl:featuremapspec type="include">
    <mpl:featurespec type="vit"/>
 </mpl:featuremapspec>
 <mpl:clipping>
     <mpl:nexusdata>
       <nsat:WKT srscode="1">
  ...textual representation of an object describing an area...
       </nsat:WKT>
     </mpl:nexusdata>
 </mpl:clipping>
</mpl:MapPL>
```

The generated feature map could look like this:

```
<mpl:featuremap>
  <mpl:feature shape="rect" coords="220,54,230,74" type="vit"
       obj="nexu://..."/>
  <mpl:feature shape="circle" coords="330,120,10" type="vit"
       obj="nexus://..."/>
  <mpl:feature shape="polygon" coords="570,265,575,280,565,280"
type="vit" obj="nexus://..."/>
</mpl:featuremap>
```

The following example specifies that the generated map should be delivered as a 4 color 160x160 gif picture using a provided palette. The features *road*, *house* and *vit* will be drawn in the colors red, blue and yellow.

```
<mpl:MapPL>
  <mpl:imagemapspec>
    <mpl:imagesize width="160" height="160"/>
    <mpl:imageencoding format="gif" bits="2" palette="provided">
      <mpl:palette>
        <mpl:rgbdef r="0.0" g="0.0" b="0.0"/>
        <mpl:rgbdef r="1.0" g="0.0" b="0.0"/>
        <mpl:rgbdef r="0.9" g="0.6" b="0.0"/>
        <mpl:rgbdef r="0.1" g="0.2" b="0.9"/>
      </mpl:palette>
    </mpl:imageencoding>
  </mpl:imagemapspec>
  <mpl:featurecolors>
    <mpl:featurergb type="road" r="1.0" g="0.0" b="0.0"/>
    <mpl:featurergb type="house" r="0.1" g="0.2" b="0.9"/>
    <mpl:featurergb type="vit" r="0.9" g="0.6" b="0.0"/>
  </mpl:featurecolors>
</mpl:MapPL>
```

The last example shows how to request a vector map:

```
<mpl:MapPL>
  <mpl:vectormapspec>
    <mpl:vectorencoding format="svg">
      ... some format specific details...
    </mpl:vectorencoding>
  </mpl:vectormapspec>
</mpl:MapPL>
```

12 Navigation Parameter Language (NPL)

12.1 Idea

In order to query the Navigation Service, the Navigation Parameter Language (NPL) was introduced. It allows to specify for instance which kind of route has to be calculated (e.g. shortest path, traveling salesman), positions on the route (e.g. start and destination), and further parameters to adapt the calculated route to the user's preferences (e.g. the desired locomotion type).

12.1.1 The Language Elements of NPL

Basic Navigation Options

shortest_route
The shortest route leading through a sequence of given positions is calculated. A fixed order of positions is given in the navigation query. These positions have to be visited in the specified order.

best_order
The optimal order of the positions specified in the navigation query is calculated (traveling salesman problem). The start and destination are specified (for a round trip the destination is equal to the start) for this query as well as other intermediate positions to be visited on the trip. The calculated route is the shortest route leading from the start through every intermediate location (in an arbitrary order) to the destination. If the traveling salesman problem proves to be very complex, also a heuristically determined near-optimal solution may be returned.

Defining Positions

start
The start element specifies the first point of the trip.

intermediate
The intermediate elements specify the points that you want to visit on the current trip.

loc_start, loc_intermediate
The loc_start and the loc_intermediate elements are similar to the start and intermediate elements, respectively (see above). Additionally, a locomotion value (see below) can be specified for the part of the trip succeeding the position defined by loc_start or loc_intermediate. This means, different locomotion values can be specified for different parts of the route. Locomotion values can only be specified for shortest route queries. If there is a start or intermediate value instead of a loc_start or loc_intermediate value in a shortest route query, then the global locomotion values defined later in the query are valid.

end
The end element specifies the last point of the trip given by a position.

position
The position is given by a NOL (Nexus Object Locator), an address, a point, or a name. An address is given by street names, house numbers, zip codes, and city names. A point specifies geographical coordinates of a position. The last possibility to specify a position is to use a symbolic name (e.g. "main station of Stuttgart").
Addresses and names have to be mappable to corresponding geographic coordinates or NOLs.

Defining Favored Locomotion Types

locomotion
The locomotion element specifies the user's favorite locomotion types or locomotion types one does not want to use. The order of the given locomotion types specifies the preference list of locomotion types for the actual navigation task (or the actual part of the trip, respectively).

loc_type
loc_type references the means of transportation. The contents of the loc_type may be any name of a class in the AWS that is a subclass of a mobile object. Thus, instances of the given class and of its subclasses are selected as transportation vehicles. If **others** is specified as loc_type, then there are no preferred means of transportations.

loc_properties
loc_properties elements provide more details about the means of transportation (e.g. average speed, maximum speed, costs per km). Constants for loc_properties will be defined in a future release of this document.

loc_use
loc_use is a boolean value specifying whether to include the current locomotion type into the user's list of favorite locomotion types or to exclude it. With loc_use it is possible to explicitly exclude locomotion types one does not want to use. To exclude locomotion types makes sense only if the **others** value is given in the locomotion list.

Other Language Elements

time
The time element specifies the point in time when one wants to start or to end the trip.

result
In the result element the result format can be defined. Currently only the point list format is supported, which is selected by specifying **pointlist**.

12.2 Examples

The example shows a shortest route query from a start point to an end point. Given are two preferred locomotion types and the starting time.

```
<npl:npl>
  <npl:query>
    <npl:shortest_route>
      <npl:start>
        <npl:position>
```

```
            <npl:point>...a NexusPointType Value...</point>
          </npl:position>
        </npl:start>
        <npl:end>
          <npl:position>
            <npl:address>...an address...</address>
          </npl:position>
        </npl:end>
        <npl:locomotion>
          <npl:loc_type>...locomotion type...</loc_type>
          <npl:loc_use>true</loc_use>
        </npl:locomotion>
        <npl:locomotion>
          <npl:loc_type>...another locomotion type...</loc_type>
          <npl:loc_use>true</loc_use>
        </npl:locomotion>
        <npl:locomotion>
          <npl:loc_type>others</loc_type>
          <npl:loc_use>false</loc_use>
        </npl:locomotion>
        <npl:time start_end="start">
          ...a NexusTimeType value..
        </time>
      </npl:shortest_route>
    </npl:query>
    <npl:result>
      <npl:format>...result format...</format>
    </npl:result>
  </npl:npl>
```

13 Navigation Result Language (NRL)

13.1 Idea

The result of a query formulated in NPL is returned as a Navigation Result Language document.

Up to now, the only alternative for specifying the result is a sequence of points. But the language can easily be extended according to the needs of applications.

13.2 Examples

```
<nrl:nrl>
  <nrl:pointlist>
    <nrl:point>...a NexusPointType value...</point>
    <nrl:point>...a NexusPointType value...</point>
    ...more points...
  </nrl:pointlist>
</nrl:nrl>
```

14 Event Information (EventInfo)

14.1 Idea

The Event Information (EventInfo) contains all the information that is relevant for the observation of an event. Most important, it contains the PredicateLocator, which uniquely identifies the event being observed. In addition, it contains the TemplateLocator, which uniquely identifies the type of event being observed, and information about when the event was registered and when it will be deregistered, unless the registration is "refreshed" before that time.

14.2 Example

```
<eventinfo:eventInfo>
  <eventinfo:predicateLocator>
    nexus:http://observerplacement5.nexus.de|urn:onEnterArea|
    0xcf2d68a644dd219d09f66deee6874a5f/
    0xab2d65a774dff19d09f66deee6874a5f
  </eventinfo:predicateLocator>
  <eventinfo:templateLocator>
    nexus:http://observers.nexus.de|urn:onEnterArea|
    0xcf2d68a644dd219d09f66deee6874a5f
  </eventinfo:templateLocator>
  <eventinfo:name>
    onEnterAreaEvent
  </eventinfo:name>
  <eventinfo:registrationTime>
    2002-01-17T23:31:14.586+00:00
  </eventinfo:registrationTime>
  <eventinfo:deregistrationTime>
    2002-01-17T23:51:14.586+00:00
  </eventinfo:deregistrationTime>
</eventinfo:eventInfo>
```

15 Event Registration Language (ERL)

15.1 Idea

There are a number of event types that are supported by the Nexus platform. For each event type an observation module has to exist that can either observe an event locally on an Event Source or, based on distributed information, on an Observation Node. New event types can be added by implementing new observation modules.

The event modules have to be instantiated with the parameters that are necessary for the observation of a specific event. The registration messages are formulated in the Event Registration Language (ERL).

Every event has a number of event specific parameters, which are defined through the event type. Each event type is identified through the TemplateLocator. In the following the events that are currently supported by the Location Service are listed together with their parameters:

- **onEnterArea** (a mobile object enters a given area): the entering object (of type **mobile object)**, and the area being entered (of type **area**). The mobile object can be specified as an object selector, i.e. the event occurs for all mobile objects that fit the object selector.
- **onLeaveArea** (a mobile object leaves a given area): the leaving object (**mobile object**) and the area being left (**area**).The mobile object can be specified as an object selector.
- **onMeeting** (two mobile objects come within a given distance): the primary object (**mobile object**) the secondary object (**mobile object**) and the meeting distance (**double**). The secondary mobile object can be specified as an object selector.
- **distPosUpdate** (a mobile object has moved farther than a given distance): the mobile object (**mobile object**) and the distance (**double**)
- **objectsInArea** (the number of mobile objects (fitting an object selector) in the area has reached a given threshold): the area (**area**), the object selector (**mobile object**) and the threshold of mobile objects (**integer**).
- **contPosUpdate** (every time interval a position update is provided): the mobile object (**mobile object**) and the time interval (**long**) in seconds.
- **contAreaUpdate** (the position of all mobile objects (fitting an object selector) within the given area are provided every time interval): the area of interest (**area**) and the time interval (**long**) in seconds.
- **registerObject** (a mobile object registers): registering object (**mobile object**) - only available for single Location Servers!
- **deregisterObject** (a mobile object deregisters): deregistering object (**mobile object**) - available for the whole Location Service
- **registerArea** (a mobile object (fitting an object selector) registers in a given area): the registration area (**area**), an object selector (**mobile object**)
- **deregisterArea** (a mobile objects (fitting an object selector) deregisters in a given area): the deregistration area (**area**), an object selector (**mobile object**)

In the future, additional event types will be added. This can even be done at runtime.

Due to the limited accuracy of the data, a probability threshold has to be defined that determines the minimum occurrence probability up to which an event is considered to have occurred, so that an event notification is sent:

- Probability Threshold: in the range (0, 100] percent

In addition to these required parameters, there are a number of mostly optional parameters, that can be specified. If a parameter is not given, a suitable default will be used. These parameters can be grouped into different categories that we present now.

Quality of Service Parameters

The quality of service parameters are optional, but they could be used to influence the observer placement in a certain way:

- Maximum clock skew: The maximal clock skew refers to the maximal time difference of any two clocks within the observation hierarchy. (Of course this is a "statistical maximum" and not an absolute bound.)
- Maximum delay: The maximal delay refers to the maximal time it takes from the observation of the occurrence of an event to the delivery of the event notification, possibly taking multiple processing steps into account.
- Average delay: The average delay refers to the average time it takes from the observation of the occurrence of an event to the delivery of the event notification, possibly taking multiple processing steps into account.
- Message loss: The message loss refers to the average rate of lost messages over a certain amount of time.
- Duplicates: Specifies, if duplicate event notifications are allowed. Duplicate event notifications could either be introduced by the underlying system or in the event observation, e.g. when a handover of an event observation between different observation nodes is performed.

Evaluation Parameters

The evaluation parameters are those that directly influence the evaluation of a predicate, i.e. they determine if an event has occurred.

- Blocking interval: The blocking interval determines the time after an event occurrence, in which the same event is not observed again. For example, after a user has entered an area, due to the limited accuracy of the position data, there may be an oscillation of the position data showing the user inside and outside the area, which could lead to a number of events being observed. With a blocking interval, this behavior can be avoided.
- Event consumption: If an event is a composition of multiple other events, i.e. an event pattern, there can be multiple occurrences of one of these events, before the occurrence of another event, e.g. a sequence A A A B. The question is what event(s) is/are observed and when the event notification is finally "consumed", i.e. that it is no longer available for future event observations. A number of different event consumption policy have been proposed: chronicle (oldest), recent (newest), continuous (all sub-events initiate a new observation), accumulative (sub-events are accumulated).

Notification Parameters

The notification parameters affect the sending of an event notification, i.e. if the notification is sent and what the content of the event notification can be.

- Sub-event parameter values: This parameter decides how the values given in the event notifications of sub-events is included in the event notification of the complex event. The general options are: do not include, include as a flat list, i.e. as a simple list of variables of the complex event, or include as a hierarchy to keep the original

structure from the notifications of the sub events. Of course, other, event-specific solutions are possible.

Management Parameters

The management parameters directly concern the management of the event observation and the characteristics of the event occurrences themselves.

- Handovers: decides if handovers of the observation are allowed
- Maximal notification rate: limits the maximal number of notifications per second
- Deregistration interval: specifies for how long an event has to be observed (soft state). This parameter is required.

Error Handling Parameters

The error handling parameters specify how to react if an error occurs.

- Error semantics: different semantics are possible: ignore, i.e. ignore the error and continue, stop, i.e. stop if an error occurs, or warn, i.e. warn the application, e.g. using a special kind of event notification

The XML schema for ERL can be found in [XML Schema Definitions].

15.2 Example

This is an example for the registration of an onEnterArea event for the mobile object Timo Heiber and the room 2.069 of the computer science building:

```
<register:predicate>
  <register:templateLocator>
    nexus:http://observers.nexus.de|urn:onEnterArea|
    0xcf2d68a644dd219d09f66deee6874a5f
  </register:templateLocator>
  <register:name>
    onEnterArea
  </register:name>
  <register:observerURI>
    http://trompete.informatik.uni-stuttgart.de:8081/soap/
    servlet/rpcrouter urn:registerOnEnterArea
  </register:observerURI>
  <register:thresholdProbability>
    0.7
  </register:thresholdProbability>
  <!-- The following parameters are specific to the onEnterArea
  event. A mobile object and the event area are specified. The
  event occurs when the mobile object enters the specified area
  -->
  <register:parameterList>
    <register:Parameter>
      <register:name>
        Entering Object
      </register:name>
      <register:type>
        MobileObjectLocator
      </register:type>
      <register:value>
        nexus:http://democlient1:8080/soap/servlet/rpcrouter|
        urn:QueryComponent|.home/
        0x43f16facb50111d7be0c080020a23ebb
```

```
      </register:value>
    </register:Parameter>
    <register:Parameter>
      <register:name>
        Entered Space
      </register:name>
      <register:type>
        StaticObjectLocator
      </register:type>
      <register:value>
        nexus:http://democlient1:8080/soap/servlet/rpcrouter|
        urn:QueryComponent|0x734beff271e711d88a9800054e433c3a/
        0x4c602221c67911d79479080020a23633
      </register:value>
    </register:Parameter>
  </register:parameterList>
  <register:qosParameterList>
    <register:qosParameter>
      <register:name>
        clock skew
      </register:name>
      <register:type>
        real
      </register:type>
      <register:value unit="milliseconds">
        100.0
      </register:value>
    </register:qosParameter>
    <register:qosParameter>
      <register:name>
        maximum delay
      </register:name>
      <register:type>
        real
      </register:type>
      <register:value unit="milliseconds">
        100.0
      </register:value>
    </register:qosParameter>
  </register:qosParameterList>
  <register:predicateManagementParameterList>
    <register:predicateManagementParameter>
      <register:name>
        handover
      </register:name>
      <register:type>
        String
      </register:type>
      <register:value>
        allowed
      </register:value>
    </register:predicateManagementParameter>
    <register:predicateManagementParameter>
      <register:name>
        deregistration interval
      </register:name>
      <register:type>
        String
      </register:type>
      <register:value unit="seconds">
```

```
          10000
        </register:value>
      </register:predicateManagementParameter>
    </register:predicateManagementParameterList>
    <register:predicateEvaluationParameterList>
      <register:predicateEvaluationParameter>
        <register:name>
          blocking interval
        </register:name>
        <register:type>
          real
        </register:type>
        <register:value unit="seconds">
          10
        </register:value>
      </register:predicateEvaluationParameter>
    </register:predicateEvaluationParameterList>
</register:predicate>
```

16 Event Notification Language (ENL)

16.1 Idea

The event notification has to inform about the specific occurrence of an event. We distinguish the individual event occurrence, the event that is being observed (there can be many occurrences of an event that is being observed) and the type of the event that is being observed.

There are a number of elements that are part of every event notification, e.g. the ID and a timestamp describing the time when the event has occurred, and there are also elements that are specific to the event type of the event being observed.

In the following the elements of an event notification are described in detail:

- NotificationLocator: The NotificationLocator uniquely identifies the event notifications, which corresponds to a concrete event occurrence. For example the onEnterArea(Tom, Office12) event that occurred at 09:30:25. In Nexus, IDs are given as NELs (Nexus Element Locators) see chapter 8 “Nexus Locators”.
- PredicateLocator: The PredicateLocator uniquely specifies the event that is being observed, e.g. onEnterArea(Tom, Office12)
- TemplateLocator: The TemplateLocator uniquely specifies the type of event being observed, e.g. onEnterArea(<mobile object>, <area>)
- Name: the human-readable name of the event type, which does not necessarily have to be unique, e.g. onEnterArea
- Location Service: specifies the service that has observed the event, e.g. the location service
- Server: specifies the server that has observed the event, e.g. the location server with the IP address 129.99.99.99.
- Counter: specifies the number of times this event has been observed so far by the given server, e.g. the 10th occurrence.
- Scope: specifies for how long the event notification is considered valid.
- Timestamp: specifies the time when the event was observed.
- Comment: optional element
- Variable List: the list of event variables, specified as 4-tuples:
  - Name: the name of the variable, e.g. Entering Object
  - Type: the type of the variable, e.g. mobile object
  - Restriction: a possible further restriction of the object, e.g. origin: Germany
  - Value: the value of the variable, e.g. Tom's NOL

The XML schema that defines the event notification can be found in [XML Schema Definitions].

16.2 Example

This is an example for a notification about an onEnterArea event:

```
</notify:notification>
  <notify:notificationLocator>
    nexus:http://locationserver3.nexus.de|urn:onEnterArea|
```

```
      0xcf2d68a644dd219d09f66deee6874a5f/
      0xab2d65a774dff19d09f66deee6874a5f/123
    </notify:notificationLocator>
    <notify:predicateLocator>
      nexus:http://observerplacement5.nexus.de|urn:onEnterArea|
      0xcf2d68a644dd219d09f66deee6874a5f/
      0xab2d65a774dff19d09f66deee6874a5f
    </notify:predicateLocator>
    <notify:templateLocator>
      nexus:http://observers.nexus.de|urn:onEnterArea|
      0xcf2d68a644dd219d09f66deee6874a5f
    </notify:templateLocator>
    <notify:name>
      onEnterAreaEvent
    </notify:name>
    <notify:service>
      Location Service
    </notify:service>
    <notify:server>
      127.0.0.1
    </notify:server>
    <notify:counter>
      1
    </notify:counter>
    <notify:scope>
      2002-01-17T20:51:14.418+00:00
    </notify:scope>
    <notify:timestamp>
      2002-01-17T18:51:14.418+00:00
    </notify:timestamp>
    <notify:variableList>
      <notify:variable>
        <notify:name>
          Entering Object
        </notify:name>
        <notify:type>
          MobileObjectLocator
        </notify:type>
        <notify:value>
          nexus:http://democlient1:8080/soap/servlet/
          rpcrouter|urn:QueryComponent|.home/
          0x43f16facb50111d7be0c080020a23ebb
        </notify:value>
      </notify:variable>
      <notify:variable>
        <notify:name>
          Entering Space
        </notify:name>
        <notify:type>
          StaticObjectLocator
        </notify:type>
        <notify:value>
          nexus:http://democlient1:8080/soap/servlet/
          rpcrouter|urn:QueryComponent|
          0x734beff271e711d88a9800054e433c3a/
          0x4c602221c67911d79479080020a23633
        </notify:value>
      </notify:variable>
    </notify:variableList>
  </notify:notification>
```

This is an example for a notification about an onMeeting event:

```
</notify:notification>
  <notify:notificationLocator>
    nexus:http://locationserver5.nexus.de|urn:onMeeting|
    0xabcd28a644dd219d09f66deee6874a5f/
    0x276765a774dff19d09f66deee6874a5f/285
  </notify:notificationLocator>
  <notify:predicateLocator>
    nexus:http://observerplacement8.nexus.de|urn:onMeeting|
    0xabcd28a644dd219d09f66deee6874a5f/
    0x276765a774dff19d09f66deee6874a5f
  </notify:predicateLocator>
  <notify:templateLocator>
    nexus:http://observers.nexus.de|urn:onMeeting|
    0xabcd28a644dd219d09f66deee6874a5f
  </notify:templateLocator>
  <notify:name>
    onMeetingEvent
  </notify:name>
  <notify:service>
    Location Service
  </notify:service>
  <notify:server>
    127.0.0.1
  </notify:server>
  <notify:counter>
    29
  </notify:counter>
  <notify:scope>
    2002-01-18T01:31:14.586+00:00
  </notify:scope>
  <notify:timestamp>
    2002-01-17T23:31:14.586+00:00
  </notify:timestamp>
  <notify:variableList>
    <notify:variable>
      <notify:name>
        Meeting Object
      </notify:name>
      <notify:type>
        MobileObjectLocator
      </notify:type>
      <notify:value>
        nexus:http://democlient1:8080/soap/servlet/rpcrouter|
        urn:QueryComponent|.home/
        0x43f16facb50111d7be0c080020a23ebb
      </notify:value>
    </notify:variable>
    <notify:variable>
      <notify:name>
        Meeting Object
      </notify:name>
      <notify:type>
        MobileObjectLocator
      </notify:type>
      <notify:value>
        nexus:http://democlient1:8080/soap/servlet/rpcrouter|
        urn:QueryComponent|.home/
        0x546a6facb50111d7be0c080020a23ebb
```

```
        </notify:value>
      </notify:variable>
    </notify:variableList>
</notification>
```

17 Geocast Communication Service

In this chapter, we describe the geographic addressing scheme, the different components, the message format, and the interface of the geocast service.

17.1 Geographic Addressing

In general, the target area of a geocast message can be addressed geometrically or symbolically [Dürr, Rothermel 2003].

Geometric addressing uses geometric figures described by coordinates in a reference coordinate system. On the one hand, geometric addressing is very flexible since all target areas can be described by some geometric figure. On the other hand, arbitrary geometric figures can have highly complex descriptions and high computational complexity. Therefore, the current geocast implementation supports polygons and circles to describe two-dimensional target areas. In the future it is planned to extend this geometric addressing scheme to so-called 2.5-dimensional figures. Such a 2.5-dimensional figure is defined by a two-dimensional figure describing the base, the altitude of the base, and the height of the 2.5-dimensional figure. These figures have small descriptions and low computational complexity compared to arbitrary three-dimensional figures, and still can approximate most target areas well.

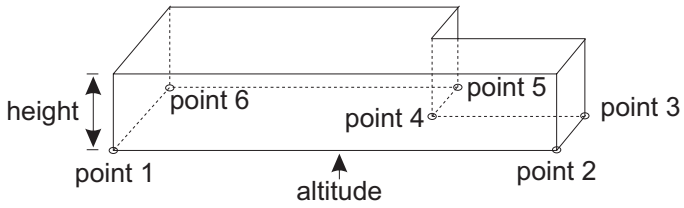

Figure 8: 2.5-dimensional Figure

Symbolic addressing uses abstract symbolic names like room or floor numbers, street names, etc. to describe target areas. These addresses are very intuitive to use since users know them from everyday life. As a drawback, symbolic addressing requires a symbolic location model defining the set of valid locations and relations between them. Arbitrary areas cannot be used. It is planned to integrate a hierarchical symbolic addressing scheme for geocast in future versions of the implementation. The current implementation only supports geometric addressing as described above.

17.2 Message Format and Service Interface

A geocast message consists of a message header followed by the message body as shown by the example geocast message below.

```
GeoApplicationSendRequest
From: Frank Duerr
From-Position: WGS84: 48.72247, 9.12445999, 0.0
Target-GeoAddress: <undefined>
```

```
Target-Area: <CIRCLE WGS84: 48.78, 9.17559866, 0.0; 52.97>
Target-Application: GeoMessageReceiver
Subject: Test
SentDate: 09.01.2003 08:29:39
ExpirationDate: 09.01.2003 08:29:39
Message-ID: 1042097398304 -1378292497
Content-Size: 42
GeoMessage-Type: PopUpMessage
<CRLF>
<Message Body>
```

The following fields are part of the message header:

- GeoApplicationSendRequest: this text identifies a geocast message sent by an application to a GeoClient
- From: the sender of the message
- From-Position: the geographic position of the sender in WGS84 coordinates. Providing this information is optional. It is so far not used by any GeoClient or message-receiving applications.
- Target-GeoAddress: reserved for future integration of symbolic addressing
- Target-Area: the geometric figure denoting the target area of the message. The following figures are supported (for backward-compatibility reasons this format is deliberately different from the WKT and GML formats used elsewhere in the Nexus project):
  - closed polygon defined by a sequence of vertices:
    <GEOPOLYGON WGS84: 48.78253484, 9.17541527, 0.0; WGS84: 48.78260221, 9.17566183, 0.0; WGS84: 48.78246747, 9.17577187, 0.0; WGS84: 48.78221839, 9.17581466, 0.0; WGS84: 48.78200403, 9.17562923, 0.0; WGS84: 48.78216531, 9.17532969, 0.0; WGS84: 48.78241847, 9.17532154, 0.0>
  - segment defined by two opposing vertices:
    <SEGMENT WGS84: 48.78247767, 9.17536026, 0.0; WGS84: 48.78204282, 9.17581466, 0.0>
  - circle defined by the center and radius of the circle:
    <CIRCLE WGS84: 48.78217756, 9.17559866, 0.0; 52.97981499757555>
- Target-Application: name of an application class to receive the message
- Subject: the subject of the geocast message
- SentDate: date and time when the message was sent
- ExpirationDate: the date and time until which the message is distributed periodically by GeoNodes in the target area. If a GeoNode receives a message with a past expiration date it still distributes the messages once.
- Message-ID: unique message identifier
- Content-Size: size of the message body in bytes
- GeoMessage-Type: application specific type of the message; can be used by the application to find out how the message should be handled.

Note: Usually, an application does not compose a message manually but uses the helper class de.uni-stuttgart.nexus.geocast.utility.GeoMessage.

Geocast messages are sent by applications to the local GeoClient via UDP datagrams. The GeoClient acknowledges a successful reception of a geocast message from the application. Note that this does not mean that the message was delivered successfully to all the GeoClients in the target area. The current implementation gives no guarantees for the delivery of geocast messages but implements a best-effort service.

Applications receive geocast messages from the local GeoClient via UDP datagrams that have the same format as shown above excluding the string "GeoApplicationSendRequest". The applications acknowledges the successful reception (this acknowledgement is only sent to the local GeoClient, which does not forward it to the sending GeoNode or GeoClient).

Before an application can receive geocast messages it has to register at the local GeoClient using a UDP datagram with the following content:

```
GeoApplicationRegisterRequest name=GeoMessageReceiver port=2260
```

"name" denotes the name of the application that is used by the GeoNode to filter received geocast messages (see above). "port" denotes the port to which the GeoClient will send received geocast messages.

To deregister, the application sends a datagram with the following content to the GeoClient:

```
GeoApplicationUnregisterRequest name=GeoMessageReceiver
port=2260
```

18 References

[XML Schema Definitions]

In order to limit the size of this document we supply the XML Schema Definitions on our website. Please visit this link to find them:
http://nexus.informatik.uni-stuttgart.de/en/research/documents

[XML Schema]

http://www.w3.org/XML/Schema

[AWS]

http://as.informatik.uni-stuttgart.de/internal/AugmentedWorldModel/AwsSchema/aws1.0

[Bauer et al 2003]

M. Bauer; C. Becker; J. Hähner; G. Schiele: ContextCube - Providing Context Information Ubiquitously. Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCS 2003)

[Bauer 2004]

M. Bauer. Event Management for Mobile Users. Technical Report 2004/02, Universität Stuttgart, Faculty of Computer Science, Electrical Engineering and Information Technology, March 2004.

[Dürr, Rothermel 2003]

F. Dürr, K. Rothermel: On a Location Model for Fine-Grained Geocast, Proceedings of the Fifth International Conference on Ubiquitous Computing (UbiComp 2003), Seattle, WA, USA, Oct 2003, pp. 18-35

[GML]

S. Cox, P. Daisey, R. Lake, C. Portele, A. Whiteside, Geographic Markup Language (GML 3.0), http://www.opengis.org/specs/?page=specs, 2003

[Hohl et al 1999]

F. Hohl, U. Kubach, A. Leonhardi, K. Rothermel, M. Schwehm: Next Century Challenges: Nexus - An Open Global Infrastructure for Spatial-Aware Applications, Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom’99), Seattle, Washington, USA, August 15-20, 1999, T. Imielinski, M. Steenstrup, (Eds.), ACM Press, 1999, pp. 249-255.

[Nicklas et al 2000]

D. Nicklas et al: Final Report of Design Workshop. Technical Report of the Research Group NEXUS, University of Stuttgart, 2000