TECHNISCHE UNIVERSITÄT MÜNCHEN

DEPARTMENT OF INFORMATICS

BACHELOR'S THESIS IN INFORMATICS

Crowdsourced IoT Data Modeling

Friederike Groschupp

Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics

Crowdsourced IoT Data Modeling

Crowdsourcing für loT Datenmodellierung

Author	Friederike Groschupp
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Dr. Marc-Oliver Pahl, Stefan Liebald, M.Sc.
Date	March 15, 2018

Informatik VIII
Chair for Network Architectures and Services

I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, March 15, 2018

Signature

Abstract

The multiplicity of devices on the Internet of Things demands the transparent orchestration of smart spaces. The Distributed Smart Space Orchestration System (DS2OS) proposes the crowdsourced creation of abstract interfaces of smart devices, called context models, to provide independence from the heterogeneity of physical interfaces. A meta model for describing context models has already been developed, but mechanisms that allow collaborative crowdsourcing are still missing. This paper develops the design of the Context Model Repository (CMR), a central server storing all valid models in use and accepting the submission of new ones.

While identifying the overall requirements for the Context Model Repository, we focus on its main responsibility: ensuring the correctness of models before they can be submitted to the CMR. A model needs to be validated and minimized to be considered correct.

Context models are described using XML. Existing standards for XML validation and minimization are not fit to interpret the semantic properties of context models. Therefore, we implemented a custom validation and minimization process for context models that is deployed in the CMR. We present the modular design of our solution. We show that our implementation fulfills the demands on the context model repository in terms of correctness, performance, and usability. With this work we lay the foundation for the crowdsourced creation of context models.

Zusammenfassung

Die Vielfalt von Geräten im Internet der Dinge (Internet of Things) verlangt nach einer Möglichkeit, viele solcher Geräte durch eine Software zu steuern. Um die Heterogenität der physischen Schnittstellen zu umgehen, wird im Rahmen des Distributed Smart Space Orchestration System (DS2OS) die Einführung von abstrakten Schnittstellen vorgeschlagen. Solche Schnittstellen werden Kontextmodelle genannt. Kontextmodelle sollen zukünftig mit Hilfe von Crowdsourcing erstellt werden. Ein Metamodell zur Beschreibung von Kontextmodellen wurde bereits entwickelt, aber eine Komponente, die Crowdsourcing ermöglicht, existiert noch nicht. In dieser Arbeit wird das Context Model Repository (CMR), ein zentraler Server, der alle verwendeten Kontextmodelle speichert und verwaltet, entwickelt.

Wir identifizieren in dieser Arbeit die Anforderungen, die das Context Model Repository erfüllen muss. Das Hauptaugenmerk dieser Arbeit liegt auf dem Design und der Implementierung der Prozesse, die entscheiden ob ein Kontextmodell in das Context Model Repository aufgenommen werden darf. Ein Kontextmodell muss valide sein und minimiert werden.

Kontextmodelle werden in XML beschrieben. Die Standards, die zur Validierung und Minimierung von XML Dokumenten existieren, genügen nicht, um die semantischen Aspekte von Kontextmodellen zu bewerten. Deswegen haben wir einen an unsere Kontextmodelle angepassten Validierungs- und Minimierungsmechanismus entwickelt, der vom Context Model Repository verwendet wird. Wir präsentieren den modularen Entwurf unserer Lösung und zeigen, dass die Implementierung unsere Erwartungen an die Korrektheit, Performanz und Benutzbarkeit erfüllt. Mit dieser Arbeit legen wir den Grundstein für die gemeinschaftliche Erstellung von Kontextmodellen.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

More and more smart devices are coming into our daily lives, and with them more and more opportunities to enrich, facilitate and enhance the day. This includes all parts of our lives: from a smart home increasing comfort and decreasing energy consumption, to factories optimizing procedures with the help of smart embedded systems, to whole cities being shaped by intelligent sensors and actuators. From your fridge ordering food, to a system of devices measuring and redirecting traffic flows without the intervention from a human. In every setting, the combination of numerous smart devices is the most effective, as you can gather as much information about the environment as possible and influence it through multiple actions.

"Ubiquitous computing" – this is the term used by Mark Weiser in 1991 in his article "The computer for the 21st century" [2] to describe the omnipresence of computational devices. In Weiser's vision, ubiquitous computing forms the base of a world in which computational devices are so well integrated into our daily life that we use them without thinking about it. All we notice is that these devices display information, influence the environment, or perform actions at the right time and place. The underlying complexity, the effort to reach orchestration goals, the connections between devices that are necessary – all this is invisible to the user.

Smart spaces – environments containing multiple smart devices – are controlled by complex systems. One of them is the *Distributed Smart Space Orchestration System* (DS2OS) [3]. The DS2OS offers services that can easily be deployed and adapt smart spaces in numerous ways. Such services customize the functionality of smart spaces for the user's needs, with very low effort on the part of the user. This mechanism is similar to the personalization of smartphones, which led to wide adoption of the devices.

These services need to communicate with smart devices in order to both control their actions and retrieve information from them. The means of this communication is determined by the interface of the smart device. The interface defines the communication protocol used, the data and its semantic value that can be retrieved, and the actions that the device is able to perform. The service controlling the device needs exact knowledge of those properties and needs to be specifically implemented for this device – one service may not be able to orchestrate the same kind of device of different manufacturers.

However, services like the ones proposed by the DS2OS are designed to be used in many different smart spaces, on many different smart devices from many different manufacturers with many different interfaces. All those interfaces would have to be considered in the implementation phase of an orchestration service to make it compatible to different smart spaces. Instead, to enable the seamless integration of services in different smart spaces, the DS2OS introduces abstract interfaces, so-called *context models*. Context models represent the logical and semantic structure of a device and make it possible for a service to access smart devices transparently. This access is provided by adaptation services which are implemented once for a device and are interfaced by context models. The orchestration service implementation now only has to consider one abstract interface for any type of device, instead of the numerous interfaces of all existing devices.

As well as proposing context models, Pahl pushed forward the idea of collaborative crowdsourced creation of context models [4]. The goal of this thesis is to enable this crowdsourced creation by designing and implementing a central repository storing all context models in use, called context model repository (CMR).

Besides storing and offering context models for retrieval, the CMR must also process context models. It must ensure that a context model is valid before it is committed and transform it to a minimized form. The implementation of this processing logic is the main task of this work.

1.1 Outline

To understand the problem domain the CMR is placed in, we analyze the DS2OS and the role the CMR plays in the framework in the next chapter. We see how context models are used for smart space orchestration and infer the capabilities that the framework managing context models should provide. We analyze the information model that is used to describe context models precisely, as this knowledge is the base for the design and implementation of the CMR. Concluding we will identify the requirements for the CMR.

In Chapter 3, we identify work related to our research. We discuss current approaches to semantic modeling and ontology building and point out why these approaches do not suffice our requirements. We compare our work to other projects aspiring to structure and standardize their domain information and providing a framework for community-driven content creation.

In Chapter 4, we discuss the general design of our solution. This includes the back-end design and the procedures necessary to validate context models and conform them to be stored in the CMR. We point out alternatives to our approach together with the reasons why we chose the present design.

Chapter 5 is a brief overview of the key points of the implementation. This includes external frameworks that are used as well as the transformation of the information model into a representation that is automatically validatable.

In Chapter 6, we evaluate our implementation. We show that our implementation accepts only valid data models and that its performance complies with the requirements. We also evaluate the feedback returned to the developer for usability and helpfulness.

We conclude in Chapter 7, and propose future work based on our results.

Chapter 2

Analysis

In this chapter we will define the research goals of this work. After doing so, we will analyze the general problem domain and define important terms in the realms of the Internet of Things in Section 2.2. We present the Distributed Smart Space Orchestration System and the key components relevant for this work in Section 2.3. The most relevant component, the information model that is used to define context models, is presented in Section 2.4. Section 2.5 discusses the crowdsourced creation of context models and the role the CMR has to fulfill. Finally, we identify the requirements for the CMR in Section 2.6.

2.1 Goals

To enable the crowdsourced creation of context models as proposed by [4], the context model repository is required as a central component accepting and storing the context models created. The goal of this thesis is to design and implement the basic functionality of the CMR.

This task requires several steps:

1. Analyze how the CMR is embedded in the context of DS2OS.
2. Analyze the properties of the VSL information model and the rules that context models need to comply with.
3. Identify the requirements of the CMR.
4. Design and implement an automated validation process for context models.
5. Design and implement an automated minimization process for context models.

With the information obtained in step 1 and 2, we are able to formulate the requirements in step 3. We will see that one of the main purposes to be fulfilled by the CMR is the validation and minimization of context models. The design and implementation of those processes is the main contribution of this work.

2.2 The Internet of Things, Smart Devices, and Smart Spaces

Defining the term *Internet of Things* (IoT) is not a trivial task. In 2015, IEEE proposed the following definition for small environment scenarios [5]:

> An IoT is a network that connects uniquely identifiable “Things” to the Internet. The “Things” have sensing/actuation and potential programmability capabilities. Through the exploitation of unique identification and sensing, information about the ‘Thing’ can be collected and the state of the “Thing” can be changed from anywhere, anytime, by anything.

In this thesis and in the scope of the embedding project, the Distributed Smart Space Orchestration System (DS2OS) [3], such a “Thing” is called smart device. With its features, it is possible to both sense and affect the environment remotely, integrating the physical world into computer-based systems.

An environment containing smart devices that interact with each other and offering to control those devices via software is called *smart space.* Smart spaces can be implemented in various settings, for example offices, homes, or public spaces. Within those smart spaces, smart devices provide extra functionality, like automatically controlling the room temperature, locking or unlocking doors, or offering a custom lighting.

The term *smart space orchestration* refers to the management of smart devices using software with a particular goal in mind. Typically that goal is reached by orchestrating several smart devices. The information that is required to fulfill the orchestration goal is called *context.*

An adaptation service offers access to smart devices. Through it context can be retrieved or modified. We introduce *context models*, which are used to define the structure of context. Adaptation services can provide uniform access to similar devices by using the same context model.

An *orchestration service*, or *smart service*, is implemented to fulfill the orchestration goal. Instead of addressing each device individually, which would require knowledge about the physical interfaces, orchestration services can work on the context models used by the adaptation services.

2.3 The Distributed Smart Space Orchestration System

The variety of smart devices that can either be bought or built can make smart space orchestration a complex and time-consuming task. With these problems in mind, Pahl [3] envisioned and implemented the DS2OS. The goal of DS2OS is to overcome the heterogeneity of smart devices, allow the easy orchestration of smart spaces, and encourage the development and usage of orchestration services. Besides the *Virtual State Layer* (VSL) middleware, which offers an abstraction of smart devices and their orchestration, it offers additional components like an app store for orchestration services, a directory and repository for data models, and a service management framework.

In the following, the word "developer" describes a person implementing an orchestration or adaption service or creating a context model. The word "user" refers to a person residing in a smart space and deploying orchestration services. A person can be both a developer and a user.

2.3.1 Virtual State Layer

The *Virtual State Layer* (VSL) is the centerpiece of the DS2OS. It offers various assistance: the management, acquisition, and storage of context, the abstraction of smart devices, and an overview of the devices and functionality present in the smart space. Context acquisition, which includes multiple sophisticated subtasks, is made transparent to service developers, just as context manipulation, which is done by using a set of fixed API functions offered by the VSL.

The VSL is designed as a self-managing peer-to-peer system of computing nodes, or *Knowledge Agents* (KAs). At start-up time, a service connects to one of the KAs. KAs are also responsible for communicating with smart devices and retrieving and storing context. Orchestration services are connected through the KA they are run on to the entire smart space. They can retrieve previously stored context, request current context and manipulate the environment. Services can communicate with each other by accessing each others context. The KAs of a smart space connect with one another and span the VSL overlay, which is transparent to both orchestration services and smart service developers.

Figure 2.1 visualizes the VSL's functionality as a middleware between the real and the virtual world. The real world is determined by physical features that can both be sensed and influenced by the smart devices contained in it (1). The communication between these devices and the VSL is handled by an adaptation services (2a). An adaptation service is developed specifically for each context model of a device and is specific for the interface of the device and the mode of communication it uses. It also determines which abstract interface is used with this device. Both the functionality of the device and the

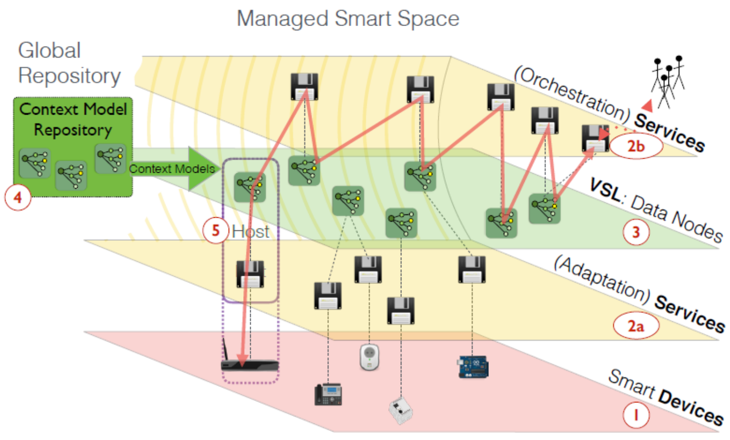


Figure 2.1: The VSL as a middleware between adaptation and orchestration services [1].

data it generates is structured by the VSL (3): devices are represented as virtual data nodes, and raw data is converted into meaningful information (e.g. the temperature scale the device works with internally is transformed to the Celsius scale). This standardized information is then used by the orchestration services (2b). The properties of the underlying hardware are not important to them.

The center of this design is data. Data needs to be structured to enable the exchange between applications. We use context models to structure data. The language used to define context models is described in Section 2.4.

The VSL offers the possibility to search for certain types of context models, allowing a service to look for possibly available context. For example, it can be checked if and how many instances of the context model "lamp" are present in the smart space.

To support portability, the points of interaction – the context models – are desired to be standardized. Only standardization of context models will abolish the heterogeneity of the hardware.

2.3.2 Smart Space Store

Considering the choice to let third-party developers provide application for smartphones as one of the factors for the success of these devices [6], the DS2OS chooses a similar approach for orchestration service creation and distribution.

The DS2OS features the *Smart Space Store*, a component similar to an app store for smartphones. Everyone can implement an orchestration service and offer the executable on the Smart Space Store. End-users are then able to easily deploy those executables in their smart space.

The *Context Model Repository* (CMR) is designed to be an independent part of the Smart Space Store. The CMR stores all context models that are in use. A smart service may only use models that are stored in the CMR. This way context models can be reused. Just like with smart services, everyone can submit context models. For a service to be accepted, all context models it references must be defined and stored in the CMR. The design and implementation of the CMR is the focus of this work. A detailed analysis of the demands on the CMR is conducted in Section 2.5.

The Smart Space Store does not only provide orchestration services and context models, but also evaluates meta data concerning the usage of smart services and context models. This data includes which context models a service uses, the relationships between services, and statistics about the quality and usage of both services and models. This data can be interesting when considering convergence mechanisms for context models (Section 2.5.4).

2.4 The VSL Information Model

The information about the environment required by services to fulfill their orchestration goal is called *context*. Context needs to be structured and represented in such a way that humans can easily understand it and machines can process it. In the VSL, context is structured by context models, which abstract the features and properties of device types. The properties and structure of context models are defined by the *VSL information model*.

To suit the needs of the DS2OS, the information model used needs to fulfill various requirements: It must be easy to understand, so that that developers can create context models without a high obstacle; it must be dynamically extensible, so that new context models can be added continuously; both its syntax and semantics must be validatable in order to provide safety; and it should be efficiently parsable.

Therefore, the VSL information model was designed as a hybrid of *Key-value pairs*, *object oriented models*, *markup schemes* and *logic based models* [4]. With the comprised advantages of those approaches, the VSL information model features:

- fast accessibility of context models through a hierarchical addressing system;
- semantics through the implicit dependencies created by namespaces;
- a typing system, which is dynamically extensible and allows computer-based validation and supports the human understanding of context;
- multi-inheritance and composition, through which dependencies between context models are represented and context model creation is facilitated.

The properties of the information model relevant for this work are introduced in more detail below.

2.4.1 Hierarchical Addressing System and Namespaces

The VSL uses a hierarchical addressing system in order to identify context models. Each context model is identified by a unique qualified address. Models are addressed similarly to files in a file system: /root/.../parent/child. This addressing system brings several advantages.

First of all, it is easy to understand, as users are used to this scheme from file systems. Additionally, the hierarchical addressing scheme gives the context models semantic meaning: Composition can be shown by aggregating several context models under the same prefix or subtree; containment is expressed by the logical tree structure. The address $smartDevice/lightSensor/irradiance$ implies that the smart device contains a light sensor, which can measure the irradiance.

This hierarchical addressing scheme automatically creates namespaces. If two different context models share a subaddress, they can still be clearly distinguished by their full address. For example, the context models $car/lamp$ and $room/lamp$ can be clearly distinguished. In the CMR, this addressing scheme divides the context models into subbranches and facilitates finding a certain type of context model, an important property for the reuse of context models.

For better readability, we will shorten the typenames of context models in the text below. However, in listings we may use exemplary, sometimes abbreviated, paths.

2.4.2 Typing, Composition and Multi-inheritance

The entities of the semantic type system in the CMR are context models. A context model has exactly one outer node, the root node. This node can contain further modes, called subnodes. Root and subnodes share the same properties in the VSL information model. Every description that is given for a "node" applies to them.

Nodes can carry a value, contain subnodes and specify meta information restricting this semantic information. It can also be defined which access groups are allowed to access their context.

In the VSL information model, each node is of one or multiple types. When a new context model is added, it automatically defines a new type that can be reused. This type is identified by the fully qualified address of the context model. This implements a dynamically extensible type system.

Using types has two incentives. Firstly, types constrain the values a node can adopt, as well as if and what kind of subnodes they can contain. This gives each context model a semantic meaning that can be computer-validated during both context model creation and runtime. Secondly, types support the general understanding of the functionality of a node by adequate naming. A node of type "light sensor" reveals its rough properties without requiring a closer look at its specification.

All types in the type system must be derived from at least one of the four basic types. The four basic types are:

- *Number*: The value of nodes of type number is a signed integer.
- *Text*: The value of nodes of type text is a string.
- *Composed*: Nodes of type composed can contain other nodes, called subnodes or elements. Subnodes are also of at least one type and have the same properties as nodes. The number and type of those subnodes is predefined by the specification of the context model and cannot be changed during runtime.
- *List*: Nodes of type list can contain subnodes as well. In contrast to the composed type, the number and type of subnodes can be changed during runtime. List is the only type that can adapt its structure during runtime.

Note that nodes can only specify values if they inherit from number or text types, and they can only contain subnodes if they inherit from composed or list types. In the following, those basic types are referred to as "basic/[type]".

We will use examples to support the understanding of the properties of the VSL information model. For this, we will use a very simple syntax, which makes use of key-value pairs. In Listing 2.1, for example, "type" and "value" are keys.

Listing 2.1 shows the definition of a simple context model which consists of only one node, the root node. The root node is of type "basic/number" and specifies the value "42". If we add this context model under the name "myNumber" to the type system, we can reuse it to directly define a new node with this properties.

```
type: basic/number
value: 42
```

Listing 2.1: Definition of a simple context model. The context model's root node is of type "basic/number".

Listing 2.2 shows a context model that contains three nodes, the root node and two subnodes contained by it. When a node contains subnodes, they are structured in an ordered list. The subnode itself is of type "basic/number" and may therefore contain a value. Note that subnodes need to be clearly identifiable, consequently they are assigned a tag which is unique in the containing node. This name does not influence the node's properties.

```
type: basic/composed
subnodes:
      |-tag: subnode1
        type: basic/number
        value: 12345
      |-tag: subnode2
        type: basic/text
        value: thisIsAValue
```

Listing 2.2: Definition of a simple context model. The context model's root node is of type "basic/composed".

We will extend the expressiveness of the VSL information model for more powerful type creation. This is achieved by introducing different actions that are described below and can be performed on existing types to create new ones.

The existing entities of the type system are reused to create new types. New types can be created through subtyping, (multi-)inheritance, and composition.

With *subtyping*, new data types can be defined by altering restrictions (2.4.3), information that limits the possible values and subnodes of a node. For example, a boolean type can be defined as being of type *number*, but only allowing the values "0" and "1". Listing 2.3 defines this type "boolean". Furthermore, subtyping can be used to give a semantic meaning to nodes. The switch of a lamp can be implemented as a node called "isOn", which in turn directly inherits from the previously defined boolean type. The result is a new type with the same properties as boolean, but with a new name describing the specific functionality. This type can still be accessed through the types from which it is derived. A service that does not know the type "boolean" can still access the instantiated node as being of type "basic/number".

```
type: basic/number
restriction: value isElementOf {0,1}
```

Listing 2.3: Definition of the type "boolean".

After type "boolean" is added to the type system, we can reuse it to define the type "switch" as shown in Listing 2.4. "switch" has the same functionality as boolean but an extended semantic.

```
type: "boolean"
```

Listing 2.4: Definition of the type "switch". It subtypes the type "boolean" without making any changes to its properties.

Composition can be used in order to create complex context models consisting of already existing simpler context models. Representing a smart device that contains a temperature sensor, an LED, and a speaker is an easy task. The developer defines a new context model called "mySmartDevice" which is of type composed and contains three subnodes: "temperatureSensor", "led" and "speaker". For this implementation, it is of course necessary that the three context models used are already defined and stored in the CMR.

To continue our example, we define the type "lamp" in Listing 2.5. The root node is of type "basic/composed". It contains a subnode which is named "switch". Note that the name has no influence on the node, we might as well give it a random identifier. What determines the properties of the node is the type. Through it, we can determine that the node is a "number", a "boolean", and has the semantics of a "switch".

```
type: basic/composed
subnodes:
           |-tag: switch
              type: switch
```

Listing 2.5: Definition of the type "lamp". The root node contains a subnode of the previously defined type "switch".

Multi-inheritance is the most complex concept here. It is necessary as one component in smart environments can have diverse functionality, and this diversity needs to be represented in its type. Consider a lamp which can be switched on and off, but also offers the additional functionality of dimming the light intensity and changing the light color. In the running system, a smart service may be implemented to work with the type "dimmableLamp", but not with the type "dimmableAndColorfulLamp". Another more basic smart service is implemented to control the type "lamp", as it only wants to switch the light on and off. We want all those context models to be reflected in the context model type.

We define the new context model "dimmableAndColorfulLamp" to be of type "dimmableLamp, colorfulLamp, lamp". Again, all three referenced context models need to be stored in the CMR. The new context model now inherits all properties – default values, subnodes, restrictions and access rights – from all three context models. This happens from the rightmost to the leftmost type. As a result, default values or subnodes with the same identifier are overwritten by the leftmost context model that specifies them.

The definition in Listing 2.6 shows how easy and straightforward the definition of this complex context model is.

```
type: dimmableLamp, colorfulLamp, lamp
```

Listing 2.6: Definition of the type "dimmableAndColorfulLamp". Multi-inheritance is used to facilitate the definition.

The context model with the modelID "dimmableAndColorfulLamp" can now be accessed by smart services through all of its types in its inheritance chain (which also contains the basic type composed). For a service accessing the node as the "lamp" type, it is transparent that the context model has to offer more context than the simple "isOn"-switch. This feature represents the functional diversity of smart devices and supports portability.

During runtime, all dependencies of a context model are fully resolved. This leads to faster processing of context models.

2.4.3 Restrictions

As mentioned above, the value as well as the subnodes of a context model can be restrained by *restrictions*. Different restrictions are defined for the four basic types. A context model can specify all restrictions from the basic types it inherits from. Throughout an inheritance chain, restrictions may only be narrowed or stay the same; they may never be relaxed. This provides type safety, as a type can always be accessed as the types it inherits from.

The value of a node of type *number* is a signed integer. Its value can be limited through the restrictions "minimumValue" and "maximumValue", which define an upper and a lower bound for the value. Throughout an inheritance chain, the interval of allowed values can only stay the same or be made smaller. Now we can define the type boolean, which we have used before, with the methods used by the VSL information model as shown in Listing 2.7.

```
type: basic/number
restriction: {minimumValue:0, maximumValue:1}
```

Listing 2.7: Definition of the type "boolean" in accordance with the VSL information model.

The value of a node of type *text* is a string. This string can be restricted by a regular expression, denoted by "regularExpression". As restrictions may only be narrowed throughout inheritance, a node should only be allowed to specify a regular expression that describes a subset of the language that is described by the regular expression it inherits. However, the inclusion problem for regular expressions was shown to be PSPACE-complete [7]. The class of PSPACE-problems contains all problems that can be solved with memory polynomial in the size of the input. PSPACE-complete problems are suspected to lie outside the complexity class NP. It is no efficient method known to decide whether a regular expression is a subset of an other. Therefore, it is not feasible to validate this restriction. We will suggest a solution for this problem in Section 4.3.2.

Basic Type	**Description**	**Restrictions**
Number	value is a signed integer	minimumValue maximumValue
Text	value is a string	regularExpression
Composed	can contain subnodes; their number, type and tags is fixed by the context model specification	
List	contains subnodes; their possible types are fixed by the context model specification; subnodes can be added and removed during runtime	minimumEntries maximumEntries allowedIDs

Table 2.1: Overview of the basic types, their properties, and restrictions.

The subnodes of a *list* can be restricted in the following ways: Their minimum and their maximum numbers can be defined, as well as which types they are allowed to be of. The corresponding tags are "minimumEntries", "maximumEntries", and "allowedIDs". Those restrictions have to be met both on runtime and committing the context model. Nodes can only be removed or added, if the number of subnodes will be in the accepted range after the operation.

Currently, the type *composed* does not define any restrictions.

When a node inherits from several basic types, all restrictions of the inherited basic types can be specified. If restrictions are already defined in several types when using multi-inheritance, restrictions are merged in the order of inheritance (from right to left).

Table 2.1 gives an overview of the basic types and their properties.

2.4.4 Access Rights

The VSL implements access control that is based on *accessIDs* for read and write access of services. A service is assigned with access identifiers through the VSL certificates used for authentication.

Users can define new accessIDs. There are two types of IDs: *readerIds*, which grant the right to read certain context, and *writerIds*, which grant the right to write certain context. The defined accessIds are stored in a central database, the accessID repository. Each entry consists of a key, which is the name of the accessId group, and a textual description of the accessID.

For each subnode, the set of readers and writers can be defined. The respective tags are "reader" and "writer". When a service wants to read context, the VSL first ensures that the intersection of the node's readerIDs with those of the service is non-empty, i.e. that they share at least one identifier. Only in this case access is granted. The same applies to write access, where writerIDs are checked.

For allowing every service access to a node, the asterisk (*) is used. If the set is left empty, only the owning service may access this context. The default is public access.

2.4.5 Semantic Invariants

The definition of the VSL information model makes the semantic validation of context models possible. Every type can be traced back to at least one basic type, which makes iterative or recursive validation possible. To ensure the safety of running systems, the integrity of all context models in the CMR must be ensured. Therefore, only well-defined context models will be accepted by the CMR.

For a context model to be well defined, it must be syntactically valid and the following semantic invariants have to apply:

- Definition of types (**R1.1**):
  All types that are referenced in the context model – including the types that are specified in the context model's type, as well as those of the context model's subnodes – need to be defined and stored in the CMR. If one type cannot be resolved, the entire context model must be rejected.

- Definition of access identifiers (**R1.2**):
  All access identifiers that are defined in both reader and writer for the root node as well as for its subnodes, need to be defined and stored in the access ID repository. If one identifier cannot be resolved, the entire context model must be rejected.

- Restrictions (**R1.3**):
  Restrictions have to fulfill several restraints that must in turn be fulfilled either in a single context model itself or between several context models when inheritance is used:

  – Only restrictions that are allowed for the type: A node may only specify restrictions if it inherits the right to define them from the basic type.

  – Restrictions must be valid themselves: Restrictions defining a lower or upper bound (number and list type) must leave an interval with valid numbers. A lower bound may not be higher than an upper bound. Consequently, a newly defined lower bound may not be higher than an inherited upper bound – and vice versa for the upper bound.

  – Restrictions may only be narrowed through inheritance: The set of valid values may only stay the same or be reduced. Accordingly, a newly defined lower bound may never be lower than the inherited lower bound – and vice versa for the upper bound.

- Default value (**R1.4**):
  If a default value is specified for a node, this value must match the basic type(s) of the node. It must also be in compliance with the restrictions defined for this node. A node may only specify a value if it inherits from number or text type.

- Subnodes (**R1.5**):
  Subnodes that are already defined in a parent model need to comply with the definition in the parent. Subnodes, inherited and newly specified, need to fulfill all of the above invariants as well. If the node containing subnodes is of type list, numbers and type of the subnodes need to comply with the restrictions, if specified. A node may only contain subnodes if it inherits from composed or list type.

2.4.6 Representation in Markup Scheme

Until now we have discussed the VSL information model – the design ideas, how context is structured, how it is described in the VSL, and which formal restrictions we can impose. It is necessary to express these structures with a data model, so that humans are able to create new context models that machines can interpret.

The VSL has previously only worked with a data model based on the Extensible Markup Language (XML) [8]. However, every other data model supporting the concepts of naming entities and associating properties could theoretically be used to represent the underlying VSL information model.

An XML document consists of hierarchically structured elements. An element can contain content, which can be textual information and/or further elements. Each element is identified by a tag, which is enclosed by "<>". An element can be described using further information called attributes. Listing 2.8 visualizes this structure.

```
<tag attribute1="attributeValue1" attribute2="attributeValue2">
      textualInformation1
      <elementtag1 attribute1="attributeValue3">
            textualInformation2
      </elementtag1>
</tag>
```

Listing 2.8: Basic properties of XML documents.

We transfer the aspects of the VSL information model to the possibilities that XML offers. Each node corresponds to an element, which is identified by a tag. The type, restrictions, and accessIDs of a node are described by the attributes "type", "restriction", "reader", and "writer", respectively. Subnodes are modeled as elements contained by a node. The textual information a node can hold represents its value. Listing 2.9 shows a context model definition in XML using the example of type "boolean".

```
<tag type="basic/number" restriction="minimumValue='0',maximumValue='1'">
      0
</tag>
```

Listing 2.9: Formal definition of the type "boolean" in XML.

2.5 Context Model Repository

The goal of this theses is to implement a central instance for storing all context models in use, called Context Model Repository (CMR). The CMR is a necessity for the crowdsourced creation of context models. This section discusses why the crowdsourced creation of context models is reasonable, but also which problems it creates. It further analyzes which requirements crowdsourcing imposes on the CMR, as well as methods that lead to a standardization of context models.

2.5.1 Crowdsourced Creation of Context Models

Crowdsourcing the creation of context models means that there is no single institution providing context models to be used for the implementation of orchestration services. Instead anyone can create a context model. To enable the crowdsourced creation of context models, a central server, the Context Model Repository, is required to control the submissions of context models and to store all context models in use.

Crowdsourcing allows a variety and quantity of context models that no single manufacturer, institution, or developer would be able to provide. Using a central institution to create context models would not work, as its ability to create context models would not scale with the diversity of smart devices and the speed of their development [9].

With the decision to use a crowdsourcing mechanism several problems can arise that must be considered and avoided:

1. A newly added context model may not be syntactically valid.
2. A newly added context model may not fulfill the semantic invariants (Sec. 2.4.5).
3. Several similar context models may exist to describe the same functionality.

4. Devices that share features may not be described by a common context model [1].

Problems (1) and (2) exist due to either a lack of skills or the malicious intentions of a developer. Measures taken against these problems, namely an automatic validation mechanism, are discussed in Sec. 2.5.3. Problems (3) and (4) arise due to the nature of crowdsourcing; if no convergence mechanisms are enforced, a heterogeneous set of context models will likely emerge. This is because a central force coordinating the creation of context models is missing. Ideas for the standardization of context models are presented in Sec. 2.5.4.

2.5.2 Design principles of the CMR

The CMR is conceptualized as an independent part of the Smart Space Store. It is open to the public, which means that anyone can commit a (valid) context model and everyone is able to browse all existing context models and use them in an orchestration service. An implication of this public access is that context models need to be validated before they are accepted. For efficient browsing and reuse of context models, semantic meta-information about context models is required.

The design of the VSL does not support the creation of multiple versions of context models [2]. As soon as a context model is committed and accepted as valid, it can no longer be renamed, altered, or removed. This is because a context model may be used by a service that expects the existence of a context model of its name with the specified properties. This also implies that a context model is identifiable by only its qualified address. A context model may only be committed when no context model with the same address already exists.

The CMR stores only validated context models in a minimized form (Sec.2.5.3).

To efficiently reuse context models and prevent unnecessary heterogeneity of context models, the set of context models ideally converges. A converging set of context models has two implications:

- If several context models for the same functionality exist, one should become the generally used one.
- If part of the functionality of a new context model is already defined in the CMR, this part should be reused through composition, subtyping, or multi-inheritance.

[1]For instance, all electrical devices with an on/off-switch should be findable and accessible by a service through a common type describing this functionality. This service could for example turn off all devices in a smart space. If not every device inherits from a common type (e.g. "switchable"), the service would need to search for all possible device types (e.g. "lamp", "fridge", "TV").

[2]Implicit versioning is possible for extending context models. For an existing "model1", one can create a context model "model2" which extends the "model1" by using it as type.

A converged set of context models would equal to a de-facto standardization and foster portability. It would prevent lifting the heterogeneity of physical interfaces to the layer of abstract interfaces. If this were to happen, the introduction of abstract interfaces would be useless. Therefore, the CMR should offer convergence mechanisms.

To reduce the latencies for context model requests and to relieve the CMR, every smart space contains a *Site Local Model Repository* (SLMR). The SLMR is a cache for the context models used in the smart space. Therefore, it always contains a subset of the context models in the CMR.

2.5.3 Context Model Validation and Minimization

Deploying smart spaces requires trust in the safety and security of orchestration systems. Both aspects are important. You neither want your running system to crash, leaving all the electronic devices in your home incapacitated, nor do you want to offer potential attacker access to your smart space.

One part of providing safety is to ensure the correctness of context models stored in the CMR (**R1**). Therefore, every context model must be validated before it is accepted. This includes both syntactic and semantic validation. A context model is semantically valid if it fulfills the invariants described in Sec. 2.4.5.

The validity of context models must also be checked during runtime, when context is modified: a new value has to comply with the type and restrictions of the node, subnodes need to comply to the restrictions of the containing list. This implies that parts of the validation also have to be carried out during runtime.

Two standards exist for the validation of XML documents: the Document Type Definition (DTD) [10] and the XML Schema Definition (XSD) [11]. DTD is embedded in the XML specification and can be used to define the structure of an XML file by markup statements. Those markup statements define a list of legal attributes and elements. The later introduced XSD, a recommendation by the World Wide Web Consortium, serves the same purpose but is more powerful. It is name-space aware, provides data typing and is able to constrain the occurrence of elements.

However, neither DTD nor XSD are fit to validate all aspects of the VSL information model [12]. Values can not be restricted by the element's attributes as it is done in the VSL information model. Inheriting information from other documents is also not considered. Consequently, we need to develop our own validating mechanism that is able to determine whether a new context model complies with the semantic invariants. XSD may be used to determine syntactic validity and that only defined attributes are used when context models are represented with XSD-verifiable XML.

In order to store context models in a concise, standardized form and to reduce the memory required, the CMR should minimize a new context model after validating and before adding it (**R2**). Sometimes, a developer may not be aware that he specifies the same information to what is already inherited. Minimization means removing all redundant information from the new context model. Redundant information describes all information that is already contained in the types that the context model inherits from and that is not altered.

A running system does not need the semantic information contained in a type system but requires fast processing and validation. Therefore, context models in use are stored not in minimized format but fully resolved by the local infrastructure. Storing a fully resolved context model omits the necessity to resolve dependencies, which increases performance in the more time-critical deployment in smart spaces.

2.5.4 Standardization of Context Models

The introduction of abstract interfaces in the DS2OS has one main goal: eradicating the heterogeneity of smart device interfaces, which prevents portability. Portability is the independence of an orchestration service implementation from a specific device interface. Portability can be realized by describing functionality instead of physical properties. However, it is important that several devices are described by the same context model; otherwise, the heterogeneity present at the device level would simply be lifted to the level of abstract interfaces.

Consequently, the abstract interface – the context model – should be standardized for a type of functionality. Standardization for context models is needed.

De-facto standardization of interfaces by vendors is likely when a few manufacturers dominate a market. The major producers introduce a standard, which is then adopted by smaller manufacturers. This standardization process is common in the network management domain, where functionality of elements is conformable and few vendors are present. In contrast, the diversity of functionality and design of smart devices is much higher. Devices are offered from many different producers and can even be easily built by a user. Therefore, it is unlikely that de-facto standardization of smart device interfaces by vendors will emerge.

Standardization driven by a central force, a standardization organization, usually takes some time and is not updated in short intervals. This does not scale given the variety of smart devices and the speed of development of new devices. A lot of device types would probably not be covered under such a standardization regime.

Pahl [4] proposes a collaborative user-based crowdsourcing mechanism. The CMR realizes an ontology of context models. Current ontology-building mechanisms require experts as contributers, manual intervention, or offline discussion. Those requirements contradict the aspiration of the DS2OS to be open to the public, to empower users all over the world, and to scale for the various applications of smart devices. Therefore automatic mechanisms supporting the convergence of context models are required. Such mechanisms are required to be implemented in the CMR (**R5**). We will discuss which problems can occur using the example of smart lamps. Then we will propose mechanisms that may solve them.

Consider the following scenario: Developer A wants to implement a service for a dimmable lamp. A dimmable lamp can be switched on and off and the intensity of the light can be set as a percentage. He defines a context model "dimmableLamp" to describe the functionality of the dimmable lamp:

```
<model type="basic/composed">
      <isOn type = "basic/number"
            restriction="minimumValue='0', maximumValue='1'" />
      <dimmed type = "basic/number"
            restriction="minimumValue='0', maximumValue='100'"/>
</model>
```

Listing 2.10: First definition of "dimmableLamp".

This context model comprises some disadvantages that damp its usability and obstruct a reasonable ontology design. The context model does not reflect the inheritance of properties; an instantiation of this context model can not be found and used as a switchable device or a lamp. The subnodes it contains, "isOn" and "dimmed", also correspond to general types and should not be defined by directly subtyping a basic type. Using the types "boolean" and "percent" would be a better choice, "switch" and "dimmable" (which inherit from "boolean" and "percent") an even better one. These better reflect the semantic meaning of the context model, facilitate the creation of context models, and support the reuse of context models.

A better definition would be:

```
<model type=".../myModels/lamp">
      <dimmed type = ".../myModels/dimmable"/>
</model>
```

Listing 2.11: Improved definition of "dimmableLamp".

Where the referenced context models are defined as:

```
<lamp type=".../myModels/switchable" />
```

Listing 2.12: Definition of a lamp which is subtyping type "switchable".

```
<switchable type="basic/composed">
      <isOn type = ".../myModels/switch" />
</switchable>
```

Listing 2.13: Definition of type "switchable" which describes devices that have an on/off-switch.

```
<switch type="derived/boolean" />
```

Listing 2.14: Definition of an on/off-"switch".

```
<dimmable type="derived/percent" />
```

Listing 2.15: Improved definition of the property of being dimmable.

It is important that the definition is divided into these parts when a new context model with similar or extended functionality is defined. Developer B wants to implement a service for a color-changing lamp, which can be set to a custom color in addition to being switched on and off. With the context model presented in Listing 2.10, there is no way to define a context model that expresses the relationship between those two kinds of lamp. With the improved definition of Listing 2.11 and the existence of the referenced types, the context model "colorfulLamp" can be defined as:

```
<model type=".../myModels/lamp">
      <color type = "/basic/text" />
</model>
```

Listing 2.16: Definition of "colorfulLamp".

A real abstraction of interfaces is only accomplished in the second case. In the first case, the heterogeneity of interfaces persists. In order to reach standardization, the DS2OS should support this style of context model creation.

Problems also arise if developers create different but similar context models for the same type of functionality. This could arise due to different naming conventions or the use of a different order of subnodes. A service working with lamps would then have to use the context models "lamp", "lamp42", "lampe", and all other context models describing a lamp. It is desirable that one of these context models for a lamp turns out as the best one and that it is used for service implementation as well as a base for extended lamps, such as a dimmable lamp.

As context models may not be renamed, altered, or removed from the CMR, mechanisms enforcing the convergence of context models must take place before adding the context model to the CMR. This means that developers should be encouraged to reuse context models instead of creating a new one and to extend already existing functionality.

A first step to reach this goal would be the introduction of a *semantic tagging* system. With such a system, the semantics of a context model could be described with semantic tags added by users on or after the commit. A context model for a lamp could be tagged, for example, with the tags "lamp", "lighting", "light", "room", and "home". This can support finding the context model when a developer searches for a certain type in the CMR and would lead to developers reusing context models more often.

Another approach is the *rating* of context models based on public statistics. A context model's rating can consist of an explicit and implicit rating. Explicit ratings can be determined by feedback received for services and context models from users and developers. Better feedback leads to a better rating. Implicit ratings can be derived from the usage of context models: an implicit rating is determined by the number of services the context model is used in, and how often those services are deployed in smart spaces. The more often a context model is used, the better its rating. Additionally, smart spaces can send error reports, if users agree.

Since a context model's ratings will be published and publicly visible, developers can surmise from the popularity of a context model how often it is already used. As using a popular context model for the implementation of a new adaptation service increases the interoperability of a new smart device, developers will most likely choose a highly rated context model. Knowing that the popular context model is compatible to a high number of devices, the developer of an orchestration service will prefer to use the most popular context model. This loop leads to de-facto standardization, as in the end the most popular and most used context model will emerge.

A third approach to reach standardization of context models would be to use *syntactic and semantic matching* and intervening in the creation process. A new context model using similar naming to an existing context model, can be an indicator for overlapping functionality. The same thing applies when a context model defines nodes with similar restrictions or subnodes to an already existing context model. For example, most numbers with the value restricted to either '0' or '1' have the semantic of a boolean. This method could be used on commit, where the developer can be informed that similar context models or functionality already exist. This approach should push the developer to rethink and improve his design.

A study on the app economy for smartphones shows that those mechanisms realize convergence of apps [6]. It also shows that the usefulness of development tool has great impact on dedication of developers, as well as the revenue that can be obtained from an application. These findings should transfer to the development of smart services and consequently also on the development of context models.

2.6 Requirements Analysis

For implementing context model creation, management, and usage as it is envisioned for DS2OS and described in Section 2.1, the framework that we propose for the creation of context models should provide the following capabilities:

- **C1**: encourage the crowdsourced creation of context models and make it possible for anyone to create context models.
- **C2**: use an information model with a corresponding data model that is easy to understand and use.
- **C3**: all context models in the DS2OS ecosystem need to be valid.
- **C4**: context models need to be consistent over time and (smart) space.
- **C5**: standardization of context models shall be achieved by convergence mechanisms without manual intervention or central guidance.
- **C6**: reduce workload by enabling the reuse of existing context models for the creation of new ones.
- **C7**: build an ontology to structure information about the domain.

Therefore, it was decided to introduce a central component storing all context models and managing context model submission, retrieval, and management. We have presented in which environment the CMR is deployed and what purpose it should fulfill. We will now identify the functional and non-functional requirements with which the CMR should comply.

Figure 2.2 visualizes how the CMR might be used in the future and what its interactions would be. A developer creates a context model and issues a request to add it to the CMR. The CMR validates the proposed context model. If it is valid, the context model can be added (**R1**) after being minimized (**R2**). A success notification is returned. For a convenient developing process, feedback about the validity of a context model should be given almost instantly (**R6**). If a context model is invalid, the error message should explain clearly why an error occurred, enabling the developer to resolve the problem (**R8**). Models shall be reused, so the CMR must offer the possibility for developers to browse existing context models and search for specific ones (**R3**). To further foster reuse, mechanisms in the CMR should be applied that lead to standardization of context models (**R5**). To encourage developers to contribute context models to the CMR, committing context models must be intuitive, just as browsing context models needs to be in order to encourage the reuse of context models (**R7**). Context models are used by services which are deployed in smart spaces. It must respond to requests for context models from the SLMR residing in the smart space (**R4**).

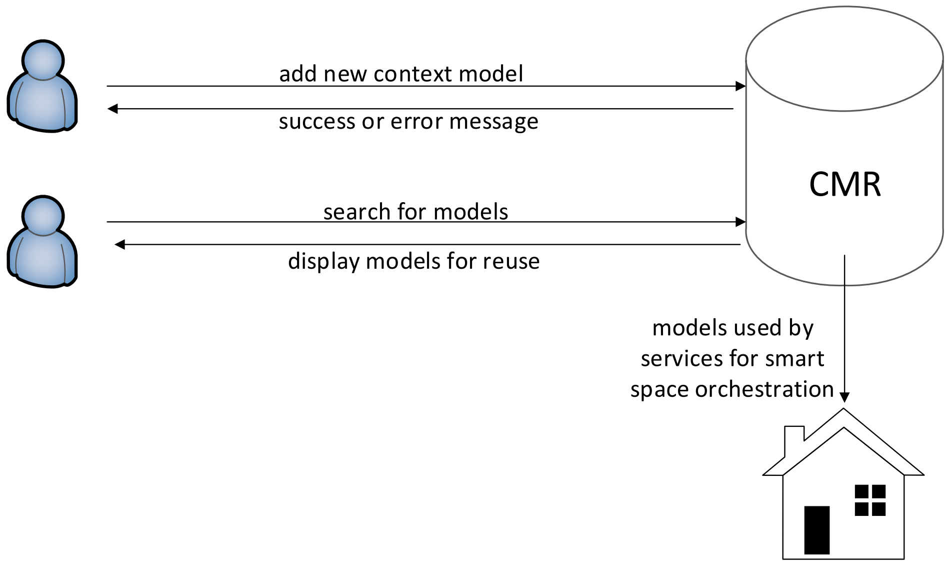


Figure 2.2: Usage scenarios for the CMR.

From the background analysis and the usage scenarios above, we can deduce the following requirements for the design and implementation of the context model repository.

Functional requirements:

- **R1**: all context models stored in the CMR must be well defined. This means that they are syntactically valid and comply to the invariants of the information model.
  - R1.1: all types referenced in the context model must exist in the CMR.
  - R1.2: all accessID referenced in the context model must exist in the AccessID repository.
  - R1.3: restrictions of a node may only be narrowed in comparison to inherited ones.
  - R1.4: the value of a node must comply with type and restrictions of a node.
  - R1.5: the subnodes of a node must be valid.
  - R1.6: the subnodes of a node must comply with type and restrictions of a node.
  - R1.7: the combination of types for multi-inheritance is only allowed if the resulting context model is valid.
- **R2**: all context models are to be stored in a minimized format.

- **R3**: it must be possible to browse and search the context models.
- **R4**: it must be possible to submit and retrieve context models.
- **R5**: mechanisms shall be deployed that lead to convergence and thus standardization of context models.

Non-functional requirements:

- **R6**: the validation process shall allow interaction (performance).
- **R7**: committing, browsing, and reusing context models shall be easy and intuitive (usability).
- **R8**: if a context model is invalid, a detailed error report shall be given. It shall be easy to identify the problem and its source in the inheritance chain (usability).

Chapter 3

Related Work

In this chapter, we identify and evaluate concepts and projects that want to solve similar problems as we do, face similar design decisions as we do, or chose to implement a similar kind of repository.

Implementing a central CMR that contains context models describing the entities of smart spaces automatically realizes an ontology, defined in Section 3.1, for smart spaces. With this in mind, we will have a closer look on collaborative ontology design mechanisms in Section 3.1. We will also introduce the basics of two Semantic Web standards: RDF and OWL.

We introduce Project Haystack in Section 3.2, an initiative in the Building Automation domain, that has the same goal as we do: streamlining working with IoT data by standardizing semantic data models. However, a different approach was chosen: Project Haystack provides a tagging system that can be used to annotate smart devices and their environments. This tagging system is built by consensus of the Haystack community. Comparing the DS2OS approach with Project Haystack will give us insights on how to enhance the usability of the CMR's type system.

We will evaluate the approach of CellML, an initiative with the goal to facilitate the exchange of biological models, in Section 3.3. Even though this initiative is rooted in a different domain, they chose a similar concept for convenient exchange of models: a central repository to store and maintain models. They offer the transformation into different data models and offer their content on a website with a structure that can also be used as an interface for the CMR.

3.1 Ontologies

The term *ontology* generally refers to an explicit specification of a conceptualization [13]. A definition adapted for the field of computer science is [14]:

An ontology is a formal, explicit specification of a shared conceptualization of a domain of interest.

Conceptualization is an abstract view of the part of the world that we define as our domain of interest. Ontologies are used to overcome the problem of implicit information by explicitly describing the conceptualization of the domain. This enables machines to infer the semantic meaning of raw data.

Ontologies are already heavily used in the network management domain and in the context of the Semantic Web [15]. The Semantic Web is the next evolutionary step of the World Wide Web. Its aspiration is to enable machines to infer semantic meaning of data, allowing "data to be shared and reused across application, enterprise, and community boundaries" [16]. Or, to phrase it differently, make data portable.

This "portable" data is enabled by technologies like RDF and OWL. In Section 3.1.1, we will have a short look on RDF and OWL, a data modeling language and a formal language for the description of ontologies, two standards commonly used for semantic modeling not only in the IoT domain.

The aspiration of the CMR – crowdsourcing the creation of context models – corresponds to te collaborative building of an ontology. Features of approaches for collaborative ontology design are shortly discussed in Section 3.1.2.

3.1.1 RDF and OWL

The Resource Description Framework (RDF) [17], is a data model for the description of resources. While RDF was generalized for a resource to be anything, within the context of the semantic web relevance is given to web resources. A resource needs to be identifiable with an Universal Resource Identifier (URI). RDF is a universal, machine readable format that is mainly used for data integration.

The main building block of RDF are statements, subject-predicate-object triples that describe resources. Those statements represent structured graphs, where subjects and objects correspond with vertices and predicates with directed edges. A subject is always a resource. An object may be either a resource or a literal that denotes a value.

Many different representations for RDF do exist, such as Turtle [18], a human-friendly format, RDF/JSON [19], or RDF/XML [20], which are based on JSON and XML, respectively. We will use RDF/JSON to formulate the above-discussed type "derived/boolean" in Listing 3.1. "prefix" could, for example, be replaced with "http://www.ds2os.org/models". The prefix of the identifying URI is omitted for the sake of readability below.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
      <rdf:Description rdf:about="prefix#derived/boolean">
            <prefix#inherits>
                  rdf:resource="prefix#basic/number"
            </prefix#inherits>
            <prefix#minimumValue> 0 </prefix#minimumValue>
            <prefix#maximumValue> 1 </prefix#maximumValue>
            <prefix#defaultValue> 0 </prefix#defaultValue>
      </rdf:Description>
</rdf:RDF>
```

Listing 3.1: Definition of the type "derived/boolean" using RDF/XML.

The first line is a reference to the standard "W3.org" namespace. It declares that this file is a RDF document. With the tag `rdf:Description` we state that we want to describe a resource identified by the URI "http://www.ds2os.org/models#derived/boolean". The elements contained represent the statements about the resource, where the predicate is denoted by the tag name and the object by the XML-value. Note that a value can be both a literal and a reference to another resource, as we used it to specify the inheritance predicate.

The difference between Listing 3.1 and Listing 2.9 depicts a huge advantage of the VSL information model: it is tailored to the application, the "predicates" describing a context model are incorporated in the information model which makes their representation more concise. In addition, prefixes are not required as the namespace is clearly fixed in our case.

Multi-inheritance can theoretically be expressed with RDF using the inheritance predicate multiple times. However, the order of inheritance, which is important to determine the final value or the order of subnodes, cannot be preserved.

The Web Ontology Language (OWL) [21] is a family of languages for the description of ontologies. It is designed to represent complex knowledge about objects, groups of objects, and relationships between objects. Built upon the RDF standard, it introduces a formal way to define evolving class hierarchies, or taxonomies. The OWL specification defines several different dialects, which all choose a different trade-off between expressiveness and decidability.

OWL defines classes, instances, and properties. A class represents a certain concept and can be described by properties. An instance is an individual from none, one, or several classes. A class may be a subclass from any other class, inheriting all properties of the parent class. All classes are subclasses of `owl:Thing` and are subclassed by `owl:Nothing` in order to allow assertions for all or no instances. Individuals can either be explicitly assigned to a class, i.e. by saying that X is an instance of Y, or implicitly, i.e. by saying that X has property Z and all individuals with property Z are instances of class Y.

Properties are either a "data type property" or an "object property". Datatype properties are a relation between an instance and an RDF literal. Object properties are relations between two instances.

With OWL, it is possible for a machine to infer information about objects just as humans are able to. With this, knowledge can be dynamically applied and is not bound to predefined procedures. This can be an extremely powerful tool.

The VSL information model does not provide such powerful inference mechanisms. Through the inheritance chain it can be determined what "classes" a smart device belongs to when it is associated with a context model. Additionally it is possible with the VSL information model to infer if two types are incompatible, i.e. when one instance cannot be of both types. This is the case when those types cannot be combined for multi-inheritance, i.e. when their restrictions or their subnodes contradict each other.

3.1.2 Collaborative Ontology Design

As ontologies are becoming so large in their coverage that more than a small group of people is required to build them [22], research and industry have taken interest in collaborative ontology development. To enable and streamline this crowdsourcing task, different approaches have been developed.

[22] identifies requirements for Collaborative Ontology Development. These include:

- Integration of discussions: An ontology needs to be build by consensus. This process is not straight forward, as different opinions exist how to model a concept or even which concept should be modeled. Therefore, tools for discussions are required that are immediately connected to the context.
- User management and origin of information: users must be able to see who submitted or altered content at which time.
- Scalability: the collaborative ontology design mechanism must be scalable both in the size of the ontology and the number of contributers.

Especially the first mentioned requirement can be deemed problematic for the CMR: such a discussion process needs time. However, when a developer is in need of a context model, he wants to be able to use it quickly for the implementation of his service. Chat and discussion functions as implemented by the ontology development environment proposed by [22] could be adopted for community discussion and support.

In [23] a collaborative crowdsourcing approach is described. However, this approach requires manual intervention, which is not suitable to scale for large ontologies. In [24] the proposed approach requires offline-discussions, which does not scale. As discussed above, extensive discussions are not compatible with the demand of instant creation of context models when they are needed.

[25] points out that in collaborative information systems, where information is managed by users, finding the right balance between expressiveness and simplicity-to-use for the information model is an important factor for success. Listening to feedback, simplifying some properties while extending others should also be done for the VSL information model, where for example more concise restrictions for data could be introduced.

3.1.3 Summary

Giving data semantic information and structuring it in ontologies has become popular and widely adopted in many domains. However, the common requirements and assumptions do not all comply with the CMR's. Manual intervention or changes to the existing type system is not only undesirable but also not possible in the CMR: the existing parts of the type system may not be altered. But structuring the information contained by the CMR in an adequate ontology improves the reusability of context models and consequently supports standardization of context models. Standardization is an important part to make smart service implementations independent from device-specific interfaces.

The CMR requires a context model to support a good ontology design at the time it is committed. Therefore, mechanisms have to be deployed to remind developers that they contribute to a joint effort, prevent them from creating isolated context models only fit to their use, and push them to create context models thoughtfully. These mechanisms could be supported by the introduction of an overlaying flexible ontology. Such an ontology could for example be realized by a semantic tagging system and make use of the ontology development and evolution approaches described above.

3.2 Project Haystack

Project Haystack [26] is an open-source initiative targeting the syntactic diversity of smart devices in the Building Automation Systems (BAS) domain. Its approach is to use semantic tagging to make data self-describing, thereby making the exchange of data between software applications easier and more efficient and enabling the convenient analysis of data.

Through the collaborative work of domain experts, an extensible data modeling approach was developed, together with consensus-approved models for BAS equipment and the data they contain. Software reference implementations for the easy adaption of data tagged with Haystack descriptions are available for different programming languages. As with DS2OS, Project Haystack aims to simplify the management of context and detaches models from vendor-specific properties.

We will have a closer look on Project Haystack's meta model and discuss its advantages and drawbacks compared to the VSL information model (Section3.2.1). After this, we will describe a proposal to introduce formal description to the Haystack meta model, the Haystack Tagging Ontology (Section 3.2.3). We will summarize our conclusions in Section 3.2.4 and identify which of the capabilities defined in Section 2.6 are fulfilled by Project Haystack.

3.2.1 Project Haystack's Meta Model

Project Haystack calls its information model a meta model, as it is common in the network management domain. Haystack makes use of tags, name-value pairs that are used to describe attributes of so-called entities. An entity is the abstraction of three types of physical objects, namely site, equipment, and sensor point. Sites are defined as buildings, identifiable by a street address. Sites contain equipment, which analyzes and influences the environment. In turn, equipment consists of sensors and actuators, the sensor points. A fourth entity that is defined is the weather, which describes outside weather conditions.

It becomes clear that the definition of the Haystack meta model is closely connected to its application domain. In contrast, the VSL information model is decoupled from any domain. This may have the drawback that it is less descriptive and less intuitive to use at first. But at the same time it does not restrict developers and allows to spread the deployment of smart services in innovative applications.

Project Haystack uses different kinds of tags that define which data type the value of a tag may be. *Marker*-tags specify only a name and no value. They are used to declare the type of an entity or an "is-a"-relationship. A building can be tagged with the marker-tag `site` to indicate it is a site. *Reference*-tags (short Ref) describe a reference to another entity. The tag's value is an entity identifier with an appended '@'. Further kinds of tags exist for basic data types, like *Str* (string) or *Number*, and more elaborate data types, like *Date* or *Coord* (coordinates). Besides those scalar kinds, three kinds exist for collections, which are *List* (list of values), *Dict* (array of tags), and *Grid* (two-dimensional table).

Project Haystack offers more native data types than the VSL information model. Those types can be assembled from the basic datatypes provided but their use may not be as convenient. Offering more data types enhances the usability of a meta model. However, offering more basic datatypes in the VSL information model would make the validation of context models more complex as the validation process is rooted in the basic types. Providing a library of additional and easy-to-reuse common derived types, such as

```
id:   @whitehouse
dis:   "White House"
site
area:   55000sqft
geoAddr:  "1600 Pennsylvania Avenue NW, Washington, DC"
tz:   "New_York"
weatherRef: @weather.washington
```

Listing 3.2: Entity in Project Haystack's meta model for the whitehouse. [26]

boolean, percentage, or table, is a good compromise.

Two specifically defined tags are used for referencing and description: `id` defines a unique identifier for an entity; `dis` is used for a short but fully descriptive characterization of an entity. Note the difference to the tag-kinds above: `id` and `dis` are reserved tag names. They do not explicitly determine the data type of a tag.

Listing 3.2 shows an example modeling of the White House. The entity is defined by seven tags: `id`, `dis`, `site`, `area`, `geoAddr`, `tz`, and `wheatherRef`. `id` and `dis` are used for referencing and description as described above. `Site` has no value and is a marker-tag defining the entity to be a building. `area` specifies the entity's area as a number, `geoAddr` and `tz` determine the building's address and its timezone as a string. `weatherRef` is a reference to an entity describing the weather station for Washington, DC.

The representation of entities in the Haystack Meta Model is very intuitive and easy to read and understand for humans. It does not even require the understanding of a formal language, which the VSL information requires as it is expressed by using XML.

References can be used to describe the relationship between entities, such as containment. The core structure for containment is the above described entity hierarchy "site – equipment – sensor point". An entity can have multiple references, making it possible to define multi-dimensional tree structures. Listing 3.3 shows the entity AHU, an air handler unit. The equip tag marks it as equipment, the `ahu` tag specfies that this entity is an air handler unit. The tag siteRef, which is a reference to a site entity, expresses that the entity "@whitehouse.ahu" is inside the site "@whitehouse". The mentioned references correspond to the relationships that can be expressed in the VSL information model by inheritance and composition. However, references in the Haystack model can be defined individually and are more distinct, making the type system more expressive.

3.2.2 Tag Database

The Haystack community is working on creating a tag database that covers components of the BAS domain. Right now, this database contains around 230 tags together with

```
id: @whitehouse.ahu3
dis: "White House AHU-3"
equip
siteRef: @whitehouse
ahu
```

Listing 3.3: Entity in Project Haystack's meta model for an Air Handler Unit (AHU) in the whitehouse. [26]

their kind, a textual description, and related tags. Contrary to the deign of DS2OS, this knowledge base can be extended individually with new tags to fit application areas which require more specialized or additional tags. The Haystack framework does not provide universal orchestration services relying on consistent models.

The tags are then deployed either in end-devices or in administering Haystack servers. The ideal situation is the first case, but it requires devices that are Haystack-enabled. Tags can then be locally stored and exchanged directly between smart devices.

3.2.3 Haystack Tagging Ontology

Charpeney et. al [27] criticize Project Haystack for lacking some features that are valuable in the IoT domain. The Haystack data model exists only in textual format; no formal representation was previously introduced. This may lead to scalability issues when the number of connected devices increases. Another point is that the implementation of the API barely complies to the restraints of IoT. It requires too much computing power for many of the embedded smart devices.

Charpeney et. al aim to formally redefine Haystack tags using Semantic Web technologies. They propose the Haystack Tagging Ontology (HTO) with a new ontology design pattern that enables both the automatic-processable representation and the easy-to-use textual representation with tags. To accomplish this, the vocabulary and the ontology are split into two parts, with a common meta-model making their relationship consistent. This corresponds to decoupling the data model from the information model.

The meta model defines `Htags` and `HEntities`. An `HEntity` may have `HTags` and references to other entities. This corresponds to tags being assigned to entities and entities using references to express physical relationships in the original Haystack definition.

Each tag in the vocabulary is an `HTag`. RDF is used to represent the Haystack vocabulary in a widely adapted and formal format. This way the vocabulary can be used with existing standards and already implemented technologies. The vocabulary is aware of the presence of the domain model; every tag is associated with a component of it.

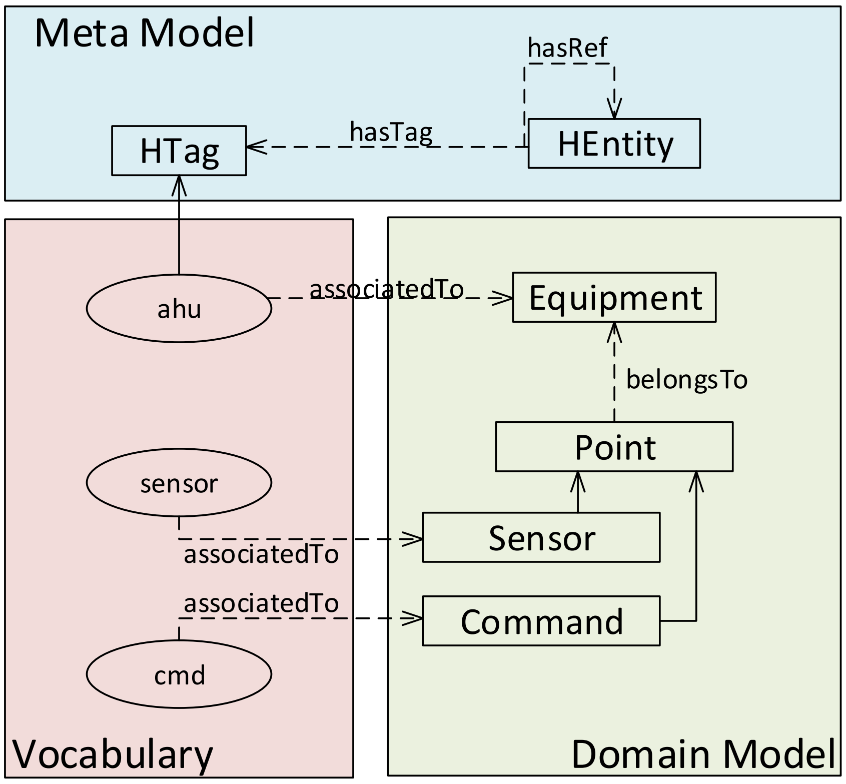


Figure 3.1: Brief concept of HTO. Only exemplary tags and an exemplary part of the ontology is displayed. For more detailed visualization of the domain model refer to [27]

To enable automatic processing, the semantic meaning behind the tags needs to be structured and formally presented. The domain model is structured by an ontology, which links the more abstract components. The central components are `point` and `equipment`. Part of the ontology is shown in Figure 3.1. Mapping tags to those abstract components has the effect that querying is simplified immensely. Now queries requesting all equipment can be easily formulated. Without the ontology, a search requires using all tags that describe equipment.

With the ontology introduced, it is also possible to detect when a model is tagged inconsistently. This occurs, for example, if two tags which should not be used together are used to describe one model. The simple tagging approach does not have this capability.

The introduction of such an ontology corresponds to the above stated idea to introduce an overlaying ontology to the CMR. Such an ontology would allow improved reasoning, make the CMR compatible with standards such as RDF and OWL, and introduce a structure that is flexible and is capable to evolve. However, in such a scenario, this additional ontology is aware of the already existing information, inverted to the design of the HTO.

3.2.4 Evaluation

Haystack introduces a tagging system for BAS applications which is easy to use and highly flexible. Tag names, kinds, and restrictions are easy to understand. In the standard definition of Haystack, no restrictions are imposed on the combination of tags and no semantic relationships besides containment are modeled. This may hinder the automatic processing of the tagged data. The proposed HTO maps tags to abstract components and structures those. In contrast, the CMR's structure of similarly defined context models in the CMR should already form an ontology and its formal information model enables automatic processing.

The introduction of an ontology that works on top of the basic meta model is also an interesting approach for the CMR. Such an ontology could be realized by tagging context models. Those tags form the ontology. The structure of this ontology can be constructed and adapted with mechanisms discussed in Section 3.1. Additionally, such a tagging system can help to understand the purpose of a context model, as it provides additional semantic information, and supports the reuse of context models, as better search functionality can be provided.

The vocabulary of Haystack tags can be customized on a per-project or per-equipment basis. This entails that the vocabulary used is not consistent in all Haystack systems. The knowledge base is stored and modified locally. In contrast, smart services in the DS2OS rely on all context models being unchanged and available. A central repository is necessary to ensure consistency.

Capability	**Project Haystack**	**CMR**
C1: Crowdsourcing	+	++
C2: Easy information model	++	++
C3: Validity of models	o	++
C4: Consistency of models	-	++
C5: Standardization	-	(+)
C6: Reuse of models	+	(+)
C7: Ontology	(+)	++

Table 3.1: Comparison how Project Haystack and the CMR fulfill the required capabilities.

Table 3.1 summarizes how Project Haystack fulfills the required capabilities we identified in Section 2.6. Theoretically, everybody can contribute to Haystack. However, the domain covered by Haystack is relatively small and expertise about equipment is required, which limits crowdsourcing (**C1**,+). The meta model used by Haystack is very easy to understand and intuitive (**C2**, ++). Models are not validated formally but specified through a consensus-building process (**C3**, o). The tag database can be individually altered, making them possibly inconsistent between different application spaces (**C4**, -). Every entity is defined only once which results in standardization, but this standardization is reached by manual effort (**C5**, -). Tags are reused and used to describe several concepts (**C6**, +). For now, it is assumed but not proven that mechanisms deployed in the CMR will have this effect. The core Haystack specification does not define a formal ontology yet but it offers the possibility to be extended (**C7**, (+)).

3.3 The CellML Project

The CellML project [28] aims to store computer-based mathematical models and facilitate their exchange. For this purpose, the initiative specified a modeling language, CellML. As the initiative is rooted in the biology domain, CellML is mainly used to describe models of physiological processes and biological structures, although its capabilities are not limited to this domain. The CellML project facilitates the exchange of biological models, as it formally describes their properties with a standardized and acknowledged data model and provides a central repository. Models are submitted to the CellML repository concurrent with the submission of a scientific publication.

To understand the environment of the CellML repository, we will introduce the modeling language CellML briefly below. In Section 3.3.2 we will discuss the CellML repository, which shares many properties with the CMR.

3.3.1 CellML Modeling Language

CellML is an XML-based language capable of describing mathematical models. CellML is capable of defining models using differential equations and linear algebra.

The main structures CellML models define are called components. A component is a functional unit that may correspond to a physical object or may be an abstraction to facilitate modeling. A component can contain variables and mathematical equations manipulating those variables and determining their relations. Components are described only by a name-attribute, while variables can be described by a name, an interface, an initial value, and possible units. CellML articulates mathematical equations by using MathML [29]. MathML is a low-level specification for mathematical content.

CellML supports the grouping of components, two predefined types are containment and encapsulation. Containment expresses that a component is physically contained by the parent component, just as it does in the VSL information model. Encapsulation is used to hide complexity of a large network of components by providing a single component acting as interface.

Metadata is used in two ways by CellML. Firstly, it may contain information about the model and its history, like the authorship, its creation date, and key words related to it. To include this information is not enforced, but highly recommenced. The only compulsory information is the citation of the published paper the model is described in.

Secondly, meta information can be used to give additional information about elements contained by a model. This way, context can be given for the variables, entities and processes described. It may also include information useful for simulation, such as optimal parameters. The use of this information supports the reuse of CellML models, as they ease the search for and they improve comprehensibility and usability of models.

Annotating CellML models with meta-information plays an essential role for the usability of a model. Users are encouraged to annotate extensively. Adding a tag system to the CMR as described in Section 3.2.4 would have the same positive impact.

3.3.2 The CellML Model Repository

The design and specification of a modeling language to enable the exchange of formal models is not enough, especially as high throughput experimental techniques produce an enormous amount of data. A centralized database is required as well. Therefore, the CellML Model Repository [30] was introduced. Besides storing and offering CellML models, the CellML repository implements two important features: the curation of models to ensure model quality and model annotation in order to simplify maintenance of models.

Models stored in the CellML Model Repository need to be in accordance with at submitted paper. To ensure the quality of models and to detect syntactical and semantic errors, the CellML Model Repository implements a curation system. Curation is designed to be a task of the community.

A star system is used to represent the current curation status of a model. A model with zero stars has not been curated yet. One star signifies that the model is consistent with the description of the published paper. Two stars denote that the model is free of typographical errors, is complete, its units are consistent, and that it is able to reproduce the results from its paper. Three stars indicate that the model satisfies physical constraints. Level-three curation is required to be done by a domain expert.

In the CMR, the submission of a context model is not bound to the submission of a scientific paper. Therefore, the expected number of context models is higher than the one for the CellML repository. Ensuring the validity of context models through a manual curation system will not scale. However, a context model being valid does not entail that it is of good quality. A quality management system similar to CellML's could be used to influence the ratings of models (compare Section 2.5.4).

To support the curation the CellML project offers editing and simulation environments. They are capable of pointing out obvious typographical errors and unit inconsistencies by error messages and of displaying the hardly human-readable format of MathML equations in an easily readable manner. If the model is able to be run, the simulation output is compared to the results of the paper. The curation process also requires to contact the authors of a paper if errors cannot be resolved or results cannot be reproduced.

Providing a customized editor for the VSL information model would serve the same ideas as these tools: Encourage participation in creating quality content that is stored in a central repository and benefits many. Reducing the work required for this content creation with specialized tools can motivate to participate.

To make it easy to find, understand and reuse CellML models, the initiative encourages its peers to use model annotation extensively. Models can be annotated with the two kinds of meta information mentioned above, which includes general information about the paper and semantic information about components or variables in the model. For annotation, the RDF standard [17] is used. To give semantic meaning to the mathematical description of the biological systems, existing ontologies and constrained vocabularies for the domain are used.

The CellML model repository offers to browse the stored models by category. In addition, it features two different kinds of searches. One takes freely choosable text as input and matches it with the textual information about the models. The other one is ontology based: one can select an ontology term and models with tags related to this term are then suggested.

Capability	**CellML Repository**	**CMR**
C1: Crowdsourcing	o	++
C2: Easy information model	-	++
C3: Validity of models	o	++
C4: Consistency of models	+	++
C5: Standardization	n.a.	(+)
C6: Reuse of models	(+)	(+)
C7: Ontology	+	++

Table 3.2: Comparison how the CellML Repository and the CMR fulfill the required capabilities.

3.3.3 Evaluation

Just as with the Context Model Repository, the creation of content for the CellML Model Repository is a community-driven effort. However, the target group of contributers differs. CellML's mathematical models are far more complex to read and understand, especially for non-experts of the domain, than the designed-to-be-comprehensible VSL context models.

An other difference lies in what kind of models can be submitted: models in the CellML Model Repository must be in accordance to a published paper, while no restraints are put on context models for the CMR.

The two differences mentioned above lead to a different approach of ensuring the validity of models in the model repository. The steps of the CellML curation process require manual work. However, the designers of the CellML deem that specialized editors are able to support this manual work. The validation process of context models can be automated. The part consisting of manual work – creating the new context model – can, however, also be facilitated with a specialized editor.

The idea of manually inspecting models can, however, be used for quality assurance of context models in the CMR. A context model being valid does not imply anything about its quality and complex self-learning mechanisms would be required to automate this process. Offering domain experts to collaboratively rate and comment context models may raise the overall quality of context models.

The design of the website of the CellML repository is a good model, as it fulfills the requirements we have identified for the CMR: browsing of models, text search, and ontology search. On the main page, several categories invite to browse models in a structured manner. Right next to it a text field offers the simple text search, while the more elaborate ontology search is linked.

Table 3.2 summarizes how the CellML repository fulfills the required capabilities we identified (compare to Section 2.6). CellML does not foster crowdsourcing so much, as

Capability	**Project Haystack**	**CellML Repository**	**CMR**
C1: Crowdsourcing	+	o	++
C2: Easy information model	++	-	++
C3: Validity of models	o	o	++
C4: Consistency of models	-	+	++
C5: Standardization	-	n.a.	(+)
C6: Reuse of models	+	(+)	(+)
C7: Ontology	(+)	+	++

Table 3.3: Comparison how Project Haystack, the CellML Repository, and the CMR fulfill the required capabilities.

it is focused on models published in scientific papers. However, it encourages scientists to submit their models and curate models of their domain (**C1**, o). Due to the more complex nature, the information model is not easy to understand (**C2**, -). Models are not validated when they are added to the repository. Their quality is rated and improved as they are already part of the database (**C3**, o). It is aspired to make the model consistent with the one presented in the paper, which is similar to ensuring the consistency of context models in the DS2OS ecosystem (**C4**, +). Currently, models or parts of models cannot be reused to build a new model but this feature is to be implemented in a future version (**C6**, +). The models themselves do not form an ontology, but they are structured with a tagging system (**C7**, +).

3.4 Summary

Evaluating current standards for ontology design, development and evolution, we concluded that many of the assumptions taken do not comply with the CMR. Creation of context models is time-sensitive and leaves no room for discussion and community consensus. Once a context model is committed, it may not be altered or removed, which makes changes to the type system impossible. To circumvent those problems, a second ontology consisting of semantic meta-information about the type system could be introduced.

We introduced two different projects, Project Haystack and the CellML Model Repository. Project Haystack is placed in the same domain as DS2OS and aspires the same goal: giving data semantics. However, the approach they chose is quite different and some of the capabilities that we deem important for our system are not fulfilled sufficiently.

The CellML is rooted in the biology domain. However, the approach they chose to facilitate the distribution and exchange of biological models resembles the CMR: A central instance storing and maintaining models together with tools facilitating the necessary manual work. However, the nature of their domain has negative impact on the fulfillment of the capabilities. CellML models are rather complex, which limits open-to-all crowdsourcing and the automatic validation of models.

With the mechanisms that are implemented in this paper, the CMR complies with most of the capabilities defined. Table 3.3 shows that with the current implementation, **C1-C4** and **C7** are fulfilled. With the implementation of the mechanisms described in Section 2.6, **C5** and **C6** are also expected to be fulfilled.

Chapter 4

Design

This chapter introduces the design chosen for the Context Model Repository. While the main emphasis is laid on the final design, possible alternatives together with their advantages and drawbacks will be discussed.

It is expected that the applications implemented will not be too time-critical. This is because the processing time for the expected size and inheritance depth of context models will be in the range of seconds, while a response time in the range of minutes would still be acceptable (**R6**), even though not desired. It is also expected that the VSL information model may be altered and extended in the future. Therefore, performance is not the most important criteria, while extensibility, maintainability, reusability, and flexibility weigh in more.

4.1 Structure of the CMR

The structure of the CMR can be divided in three parts: the storage of information, the logic for the processing of context models and access IDs, and the interface between the CMR and the developers retrieving and adding information.

This structure is pictured in Figure 4.1. It also depicts the dependencies between the components. In the future, the CMR will be accessible through a web interface, tailored for human use, and one command-line-based interface, which is also fit for communication with components like the SLMR. Through these interfaces, the set of operations offered by the CMR can be accessed. These operations are split into two sets. One comprises of simple operations which simply retrieve data stored in the CMR. The operations of the other set require more complex procedures and may also submit new content to the CMR.

The more detailed design of the individual layers is discussed below.

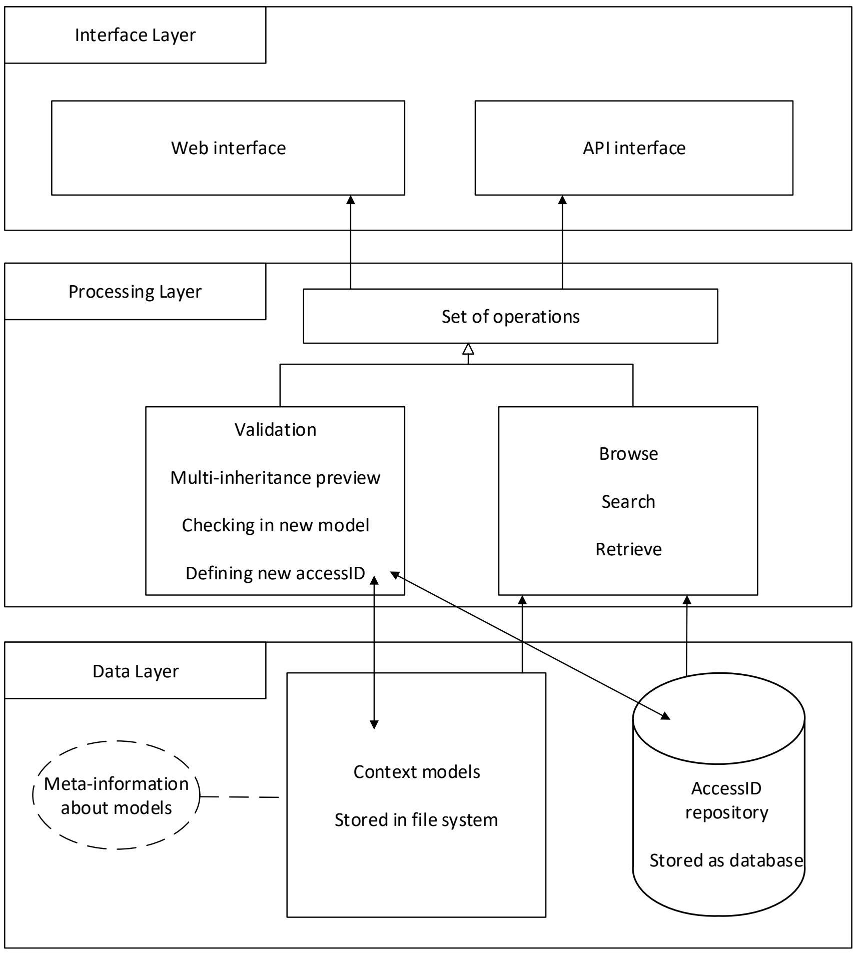


Figure 4.1: Rough structure of the CMR.

4.2 Data Layer

The data layer contains all data stored by the DS2OS: the context models, the accessIDs, and meta-information about context models and the type system.

As described above, context models are stored in text files identified by hierarchical addresses. The addresses correspond to the identifier of the type defined by this context model. Storing those files in a file system reflects this structure. It also provides good performance to access the content of the context models, as no communication with an external database is necessary and no logical tree structures need to be resolved or traversed. According to **R1** and **R2** all context models stored in the CMR are valid and minimized.

Together with context models, meta-information about them needs to be stored. This includes basic information, such as the authorship or the time the context model was submitted, as well as more elaborate information, such as semantic tags or correlations between tags exceeding the inherent structure of the type system. Such information is required to build a second, flexible ontology. This meta-information is not needed for the immediate processing of context models – validation, resolving, retrieval of context models for services – but rather supports the reuse of context models and finding adequate context models to implement a service. Therefore, this meta information should be stored separated from context models, for example in an independent database.

AccessIDs, which determine the right groups to read and write context, are an independent instance from context models. No accessID is bound to a certain context model and may be referenced by no or multiple context models. As accessIDs require separate maintenance from context models, they are stored in a separate database, called *AccessID Repository*. As of now, the only information about context models consists of the unique identifier together together with a textual description of the accessID. This information can be extended with the authorship, the submission date, and further meta-information about the accessIDs.

4.3 Processing Layer

The processing layer forms a bridge between the data layer and the interface offered to developers. Its task is to respond to requests either querying for data or aspiring to submit new data by retrieving data from the CMR or adding new one. Those requests can be split in two parts: basic requests that want to retrieve data from the CMR and more elaborate requests that require sophisticated logic.

Basic tasks include browsing, searching, and retrieving context models and accessIDs. These tasks only require to retrieve data from the CMR. Browsing context models allows to have a look at the existing type structure, traverse it, and have a look at existing context models. Searching context models offers to find context models that make use of certain terms or fulfill a certain semantic functionality. To enable both, key-word-based search functionality needs to be offered together with an ontology-based one, which in turn requires a semantic tagging system. Retrieving context models allows caching instances in a smart space to download and store context models. These context models are then used by smart services to fulfill their orchestration task.

More elaborate tasks include validation of context models, preview of multi-inheritance, and the submission of context models or accessIDs. Aside from retrieving data from the CMR, those operations require complex processing logic and may also submit new information. Allowing a context model to be validated and feedback given to the CMR enables a developer to assess the quality of his context model without having to make irreversible changes to the type system. Enabling the preview of multi-inheritance supports the creation process especially when many complex types are combined. The developer can be sure of how the final combined type looks like and whether the combination is valid or not.

The submission of a new context model combines most of the tasks above: the context model needs to be validated, which in turn requires retrieving, parsing, and resolving context models. If multi-inheritance is part of the type declaration, resolving context models and assembling one type out of many become necessary. Those procedures can be reused for the different tasks. Additional procedures that are required include the minimization of context models and ensuring that the context model identifier of the new context model is valid.

The single procedures and the reasoning behind their design will be discussed in Sections 4.3.1 to 4.3.5.

4.3.1 Parsing

In this section we discuss the parsing and transformation of context models necessary to validate and work on the logical structure of a context model independent from the underlying data model.

The current implementation of the VSL only works with the XML-based data model introduced in Section 2.4.6. With no intermediate instance, developers can create context models only with this data model. With the introduction of the CMR, such an instance now exists: context models can be submitted on every adopted data model and be processed independently of it. This requires that the CMR does not handle context models on the level of data models, but on the level of the information model. This is the first reason to transform the textual description to a logical representation with Java objects.

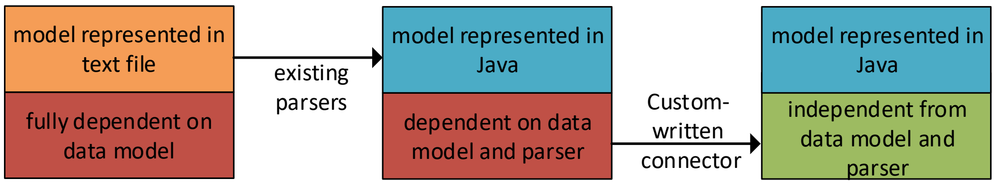

model represented in text file
fully dependent on data model
existing parsers
model represented in Java
dependent on data model and parser
Custom-written connector
model represented in Java
independent from data model and parser

Figure 4.2: Steps and artifacts in the parsing process.

The second reason is the need for a validation mechanism beyond current standards. The VSL information implements complex concepts, which include dependencies between documents, the inheritance of information, and restricting the value of nodes by attributes. We reasoned in Section 2.5.3 that the two standards for XML-validation, DTD and XSD, are not able to cover all aspects of the VSL information model. Developing a custom validation mechanism for semantics is consequently unavoidable, while the syntactical validity can be checked with standard instruments.

Such a validation mechanism needs to retrieve structured data, the minimization process even needs to manipulate it. Both need to compare different context models for equivalent data that may be defined in reverse orders. The easier processing of the information model is the third reason to abstract it from the data model.

To facilitate easy information retrieval and manipulation, the XML document needs to be parsed into an easily read- and writable data structure. Many preexisting tools exist that transform common data models into Java structures. Theoretically, one could now conveniently work on this structure. However, the makeup and properties of this structure vary with different data models and parsers.

As our goal was providing independence between the information model and data model, a second parsing step was required. The second parsing step parses the data-model-based structure to a data-model-independent structure that reflects the properties of the information model. This step, which is an easy Java-internal transformation, is the only part that needs to be implemented if a new data model is introduced or a different parser for the first step is to be used. Dividing the parsing process in those several steps provides flexibility and easy extensibility.

Figure 4.2 visualizes the design for the parsing process and the two parsing steps that are required on different layers. The first parsing step transforms the context model from a plain text file to Java objects that are easy to read and modify (orange to blue). The second step makes the information independent from the data model (red to green).

4.3.2 Validation

To comply with **R1**, every context model needs to be valid to be added to the CMR. Validation is split in syntactic and semantic validation. To discuss validation, we will at first describe our approach on how we validate context models. Afterwards, we will list accurately how the properties of context models need to be validated.

A context model is syntactically valid if it conforms to the data model it is represented in. For the XML representation it is, for example, required that every element is opened and closed by corresponding tags. Deciding if a context model is syntactically valid can be done individually for each context model, it is not necessary to consult information from the CMR for it. The syntactic validation is already provided by the parser used and thereby done during parsing. In case a context model is syntactically invalid, it is rejected and the error message provided by the parser is returned.

A context model is semantically valid if it fulfills the semantic invariants (Section 2.4.5). While it is possible to check the syntax of a context model isolated from existing context models, this does not apply to the semantic validation. In order to check if a context model complies with the invariants, one needs to retrieve a lot of additional information: which types the context model inherits from in its inheritance chain, values and subnodes it inherits, and which restrictions are handed on.

The semantic validation heavily relies on the use of context models already stored in the CMR. There are several variants to retrieve this information such that it can be used for validation. We will discuss the different approaches below. For the assessment it is important that we can assume that **R1** and **R2** are fulfilled after every submission: all context models in the CMR are valid and stored in minimized format. We will discuss two different approaches: one where all information necessary for the validation is retrieved before the actual validation, and one where the information is retrieved during the validation process.

The first approach validates the new context model against the resolved representation of its immediate parent model. The resolved representation contains all information about this context model, which includes information that is inherited and removed in the minimized information. Once this fully-resolved context model is compiled, no further context models need to be retrieved. This validation process requires an additional subtask: resolving the parent model. The rough procedure is displayed in Listing 4.1.

```
newModel = parse(id);
type = newModel.type;

parentModel = parse(type);
parentModel = resolve(parentModel);

for all information in newModel
      ensureInformationCompliesWithParentModel();
```

Listing 4.1: Outline of the first validation option in pseudo code.

The first step is to parse the new context model as described in Section 4.3.1, `id` is the identifier of the new context model. The parent model is determined by the new node's type and is retrieved from the CMR and parsed. The next step is to fully resolve the parent model as described in Section 4.3.3. Now the parent model contains all information that affects whether the new context model is valid. All information that is defined by the new context model is now validated. This includes restrictions, values, and subnodes. For example, it is ensured that restrictions are only narrowed and that a specified value matches the restrictions. If a value is not explicitly defined, it has to be ensured that the inherited value matches possible new defined restrictions. It must also be ensured that the total number of subnodes – inherited and new ones – matches the restrictions.

As only the parent model and not the new context model is resolved, subnodes that are not inherited but newly defined do not have a corresponding node they can be validated against. One option would be to add subnodes of the same name and type to the parent model and resolve them their. However, this would alter the parent model. It could not be cached and reused, as it does not comply with its original specification any longer. Therefore, we decided treat those newly defined subnodes individually and resolve them during the validation process, assigning them an individual parent model each.

The advantage of this approach is that the resolved parent model can be reused for the subsequent minimization of the context model and the resolving process can be reused for providing resolved context models that are required during the deployment in a smart space. An other advantage is that every model in the inheritance chain is fully resolved. It can be cached and reused when the same type is referenced again. In this case retrieval, parsing, and resolving do not have to be performed again which improves the performance.

The disadvantage of this approach is that meta-information is lost. If, for example, a value violates a restriction, one cannot immediately tell from which super type this restriction is inherited from. It is necessary to traverse the super nodes again. However, finding a violation of the invariants can be decoupled from finding the source of the validation, allowing for a reliable validation as well as an expandable and incrementally improvable error-reporting.

At first look, this approach may look like it imposes an unnecessary huge overhead: the whole inheritance chain needs to be traversed to the roots, which can be multiple in case of multi-inheritance, in order to resolve the parent model. However, it actually is necessary to traverse the whole chain as context models are stored in minimized format. Only in this way all basic types the context model inherits from can be identified with certainty. A little overhead is produced nevertheless: values are read and merged that may be overwritten at an higher level of the inheritance chain anyways. We do not expect that those operations do have significant overhead on the performance.

The second approach is to traverse the inheritance chain of the new context model until information is found that reveals the new context model to be invalid. A property of a model is valid when it complies with the first respective restriction in the inheritance chain or if the end of the inheritance chain – the basic types – is reached, and no violation was determined. For example, if a new node defines the restriction "minimum-Value", its inheritance chain is traversed until the first specification of "minimumValue" and "maximumValue", as they determine if the new restriction is valid. If the whole inheritance chain is traversed and no such restriction is defined, the specification of the restriction is valid when one of the basic types is "basic/number".

However, this approach requires constantly retrieving context models from the CMR while validating, making the validation process slow and not compatible with caching. Also, we would need to retrieve context models again for minimization, as the fully-resolved parent is not compiled. The presumed advantage of a better performance because not all information in the inheritance chain is considered is also void in most cases. If no restrictions are specified in the inheritance chain, the only possibility to infer the basic type(s) is to traverse until the root(s). The same holds for the definition of new subnodes.

Considering that our main aspirations for the design were modularity and maintainability, we prefer the implementation of the first mentioned method.

However, an issue concerning multi-inheritance needs to be solved when using this approach. A new context model that directly inherits from several types does not have an existing immediate parent node that can be resolved. An easy solution is to construct a temporary node that represents this parent model. This temporary context model inherits from all types in the same order that the new context model inherits from and does not specify any alterations or additions. Before the new context model is validated against it, it must be ensured that this temporary node is valid. Not all combinations of types can be used for multi-inheritance, for example if they define contradicting restrictions.

The functioning of this mechanism is visualized in Figure 4.3.

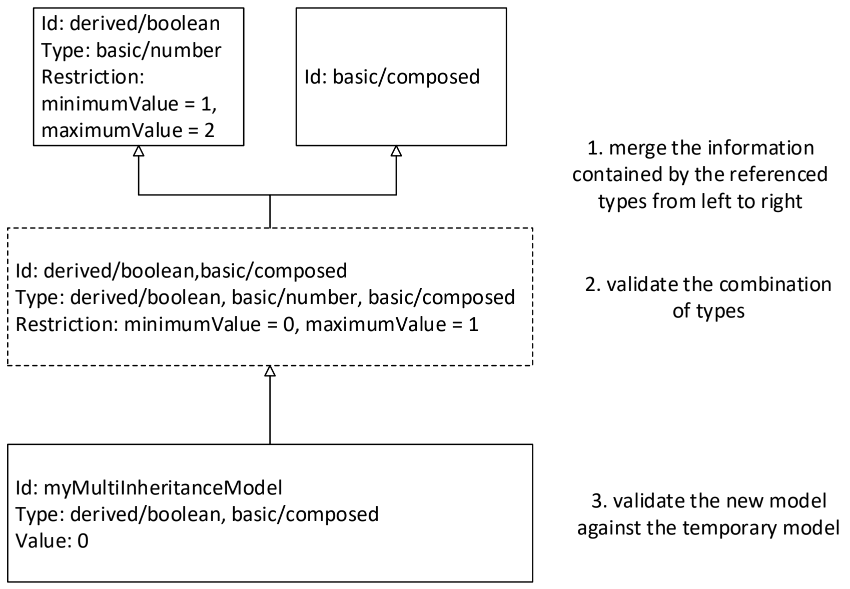

Id: derived/boolean
Type: basic/number
Restriction:
minimumValue = 1,
maximumValue = 2
Id: basic/composed
1. merge the information contained by the referenced types from left to right
Id: derived/boolean,basic/composed
Type: derived/boolean, basic/number, basic/composed
Restriction: minimumValue = 0, maximumValue = 1
2. validate the combination of types
Id: myMultiInheritanceModel
Type: derived/boolean, basic/composed
Value: 0
3. validate the new model against the temporary model

Figure 4.3: The creation of the temporary node and validation steps for the context model "myMultiInheritanceModel".

Below, we will structure the elements that need to be checked for semantic validity and structure them into groups that will serve as a template for the implementation of validators. A validator is a routine that serves to validate one aspect of the information model. Not every context model needs to be checked by every validator. Models inheriting only from "basic/number" may not define any subnodes, so we need a validator ensuring this, but the validator checking the validity of the subnodes is not required in this case. Therefore, every basic type comes with a list of its respective identifiers that are passed on to inheriting children. This also improves the performance of the validation, as only relevant aspects are validated.

Splitting the validation task into several, clearly distinguishable sub-routines makes the design very flexible. Future changes or extensions to the VSL information model can be implemented easily.

It is important to pay attention to the order in which the validators are executed: validating the value of a number node assumes that the respective restrictions are valid, just as confirming that the subnodes of a list comply with the restrictions assume that the subnodes are valid themselves.

Validation of Attributes

- *Validate Attributes:* Ensures that the only attributes defined are named "type", "restriction", "reader", and "writer". Defining an attribute with the same name twice is considered syntactically invalid and therefore caught by the first parser.
- *Validate Access IDs:* It needs to be ensured that every access ID defined in both the "reader" and the "writer" set are defined in the access ID repository.
- *Validate Type:* The type string is valid only if all the referenced types do exist in the CMR. This needs to be collated. In contrast to the other validation requirements, this is the only information that needs to be already valid for the resolving of the parent model. Therefore, this mechanism is not implemented as an independent validator, but as a part of the resolving process.
- *Restrictions*
  - *Validate Restriction Existence:* Ensures that a node only defines restrictions if it inherits from the respective basic type. For example, a node may only define the restriction "minimumValue" if it inherits from type "basic/number". It also ensures that only valid restrictions are specified, i.e. that all restriction names correspond to the six restriction types currently defined by the information model.
  - *Validate Number Restrictions:* Validates whether the newly defined restrictions "minimumValue" and "maximumValue" comply with the invariants in three ways: (1) The value of the restrictions is required to be a number represented with digits. (2) The newly inherited restriction "minimumValue" must be greater than or equal to the inherited "minimumValue" and less than or equal to the inherited "maximumValue". Vice versa for the newly defined "maximumValue". (3) If both restrictions are newly defined, "minimumValue" must be less than or equal to "maximumValue".
  - *Validate List Restrictions:* The list restrictions "minimumEntries" and "maximumEntries" need to be validated just as described above for the number restrictions. The restriction "allowedTypes" needs to be checked for the following constraints: (1) If "allowedTypes" is defined, it needs to specify at least one type". (2) The types specified need to be a subset of the inherited "allowed types". (3) If no "allowedTypes" is inherited, i.e. if the restriction has not been specified in the inheritance chain, it needs to be ensured that the specified values exist in the CMR.
  - *Validate Text Restrictions:* In the spirit of the CMR, the restriction "regularExpression" would be required to describe a subset of the language described by the inherited restriction. However, validating this is infeasible (compare

Section 2.4.3). Therefore, no validation of this restriction is performed. Instead, to provide type safety, the following policy is applied: a node inherits all regular expressions specified along its inheritance chain and the node's string value has to comply to all off them. This equals to an intersection off the regular expressions and ensures that the value fulfills all of them. With this policy, it may occur that the set of possible values is empty.

For example, we can define the type "5text" whose value needs to exist of exactly five characters and the type "numberString" whose value may consist only of digits. The value of a type inheriting from both would need to consist of exactly five digits. The same applies if one type inherits from the other.

Validation of Values

After the restrictions are validated, the value of a context model needs to be validated. If a context model defines a new value, it needs to be validated against the new restrictions if present, otherwise against the inherited ones. If no new value is defined, it needs to be checked that the inherited one still complies with possible new restrictions.

- *Value Existence*: If a node defines a value, it must inherit either from type "basic/number" or "basic/text".
- *Number Value*: If a node inherits from type "basic/number", its value needs to be a number represented with digits. Additionally, the value has to comply with the restrictions "minimumValue" and "maximumValue", if defined.
- *Text Existence*: The value of a node derived from type "basic/text" needs to match all regular expressions in the inheritance chain.

Validation of Subnodes

Subnodes require validation in two regards. They need to be valid themselves, so all of the applicable validators need to be applied to them. Additionally, their number and type need to comply to the restrictions defined by their containing context model.

- *Subnode Existence*: If a node contains subnodes, it must inherit either from type "basic/composed" or "basic/list".
- *Unique Identity of Subnodes* A tag may be defined by only one element directly contained in a node.
- *Validate Subnodes*: All subnodes need to be valid. A node can only be considered valid if all its subnodes are valid.

- *Validate Subnode Compliance*: If the node inherits from "basic/list" and defines restrictions accordingly, the subnodes need to comply to them. It is important to note that all nodes, newly defined and inherited ones, need to be considered for this. This includes checking the number of subnodes as well as ensuring that each node inherits from at least on of the allowed types.

Validation of Multi-Inheritance

When a new node inherits from several types it must be ensured that the combination of those types is valid. The combination of types is invalid if one of the following conditions applies.

- *Type Declaration*: Types must be defined in a restricted order: a type may not be specified explicitly after it has been specified explicitly or implicitly. The direction for this rule is from right to left. For example, defining a type as "derived/boolean, basic/number" is valid. "basic/number, derived/boolean" is invalid, because "basic/number" is implicitly defined by "basic/boolean" already".
- *Restrictions*: The resulting restriction combination is invalid. For "minimumValue", "maximumValue", "minimumEntries", and "maximumEntries this denotes that the lower bound may not be higher than the upper bound. The restriction "allowedTypes", created by the subset of the inherited restrictions, must specify at least one type.
- *Value*: If no value is defined by the child model, the value from the leftmost type defining a value is considered as the final value. This value needs to comply with the invariants.
- *Subnodes*: The subnodes need to comply to the restrictions: their number may not exceed the final "maximumValue" and every subnode type needs to overlap with the final "allowedTypes".

4.3.3 Resolving

Resolving context models – transforming their minimized representation into one containing all information – is not only useful for validation, but also for minimization where we will reuse the resolved parent model we created for validation. Two approaches are possible for resolving a context model: a recursive approach and an iterative approach. We will discuss the approaches below.

When a context model is resolved iteratively, the resolving process starts with the context model on the highest inheritance level and traverses the inheritance chain. In every step, all information that is not yet specified is added to the context model. This means, for example, when a new restriction type appears it is added. However, existing restrictions or values are not overwritten. This means that the context model in the next step is only searched for information that is not yet specified, which reduces effort and improves performance. However, in all cases it is necessary to traverse the whole inheritance chain to the basic types, as this is the only way to determine which basic types the context model inherits from and to resolve the complete type string. The advantage of the iterative approach is that it costs less both in regards to time and storage space.

In contrast, the recursive approach requires the immediate super node to be resolved recursively before its information is merged with the information contained by the node-to-be-resolved. This approach is probably the more intuitive one, as it reflects the mechanism of inheritance better: information is passed on to a child node that is then possibly altered or extended. Obviously the whole inheritance chain needs to be traversed for this approach. However, this also the case for iterative resolving, which ensues that the overhead created by the recursive approach is not too high in comparison.

The recursive approach offers the possibility to cache the resolved representation of context models in the inheritance chain. This is not possible with the iterative approach, because their resolved representation is never assembled. Making use of this technology would decrease costs immensely if types are inherited from several times, for example when several subnodes inherit from the same type.

Given this advantage, we decided to implement a recursive resolving process. When a node is to be resolved, the first thing that is done is to determine the immediate parent node by the node's type string. Then the context model corresponding to this identifier is resolved. In case of multi-inheritance, that is when the type string contains more than one type identifier, all context models are resolved and merged to one temporarily existing parent node. After the resolved parent node is assembled, the information of the node-to-be-resolved is merged with the parent's. This means that the type string needs to be combined and restrictions, access identifiers, or values defined by the child replace the ones defined by the parent. Subnodes are either completely inherited when only the parent defines them, inherited and merged when both context models define them, or need to be resolved when only the child defines them. This process is outlined in Listing 4.2.

```
resolve (node)
      //retrieve the resolved parent, merge information and
      //return the resulting resolved node
      typeString = node.getType();
      parent = getResolvedParent(typeString);
      node = mergeNode(node, parent);
      return node;
```

```
mergeNodes(child, parent)
      //merge the informtion contained in the two nodes,
      //child’s information overwrites equivalent information from parent
      child = mergeAttributes(child, parent);
      child = mergeValue(child, parent);
      child = mergeSubnodes(child,parent);
      return child;

getResolvedParent(typeString)
      if (typeString.onlyOneType)
            //return the resolved model with the id typestring
            node = ModelLoader(type);
            return resolve(node);
      else
            //return a node containg the merged information off all
            //specified types
            node = newEmptyNode();
            for all types in typeString
                  node2 = ModelLoader(type);
                  node2 = resolve(node2);
                  node = mergeNodes(node2, node);
            return node;
```

Listing 4.2: Outline of the resolving process in pseudo code.

The resolving process is based on the assumption that all context models are valid. This assumption is realistic, as for the resolving process only context models stored in the CMR may be considered. According to R1, all those context models are valid.

4.3.4 Error Reporting

When a context model is invalid, the context model is rejected and the developer is informed about the failure in the validation process (**R8**). The quality of the description of the error cause can range from “something is wrong”, over a more detailed description about the error type and where it occurred, to an error accompanied together with suggestions for correction. It is obvious that a developer would prefer the rightmost option, while this realization requires far more elaborate implementation.

Providing good error messages is important, not just in order to help the developer to resolve the current issue. It is also possible to foster the developer’s understanding of the rules imposed by the information model design and the specific context models he is using [31].

A good error message should follow four basic rules [32]:

- Use clear language and avoid system-internal information. If such information is required for a system manager, attach it at the end of the error message together with a note suggesting to forward this information to a system manager.

- Be precise rather than describing the general problem. In case of this application this means avoiding an error message like "Invalid value" and rather describing why the value is invalid and which node it belongs to.

- Help the developer to solve the problem. Methods like spelling correction can guess what the developer actually wanted to say and suggest the change. If the context model inherits from the undefined type "basic/numbers", the system could propose to use the existing type "basic/number" instead.

- Be polite and do not blame the developer.

Considering those rules a simple guideline for the error messages of the CMR was developed. Validation error message begins with a line stating that an error has occurred together with the unique identifier it has occurred in – the concatenation of the file name with all tags of the containing nodes – and the line in which the node is defined in the document. This simplifies the search for the problematic segment. After this, the reason for the error is given, such as an invalid value, invalid subnodes, or invalid attributes. This is accompanied by an explanation why the respective element is invalid. In most cases, more information is available: for example, if the element is inherited or which restrictions it violates. If this is the case, this information is also added to the error message.

Listing 4.3 shows an invalid definition of a context model with the id "myBoolean". When the context model is validated, it is rejected together with the error message displayed in Listing 4.4.

```
<model type="basic/composed" >
      <bool type="derived/boolean" restriction="minimumValue='5'">
            0
      </bool>
</model>
```

Listing 4.3: Invalid context model definition: the restriction "minimumValue" collides with the inherited restrictions.

```
Error at node 'myBoolean/model//bool' (line 2):
The restriction 'minimumValue' is invalid.
This node 'minimumValue': 5, inherited restriction 'maximumValue': 1.
```

Listing 4.4: Error message pointing out the identifier and the line of the invalid node, as well as the reason for the error.

A special kind of error is the combination of multiple types that are incompatible for multi-inheritance. In most cases, it is not possible to resolve this error by fixing a typo or defining a different default value. Instead, this error type reflects a more fundamental problem. To signal this, all errors of this type carry the information "multi-inheritance error" in their second line.

Besides validation errors, unexpected errors may occur during the validation process. Those errors are marked as "internal errors". Such errors may hint at bugs in the implementation or at system failures such as a broken communication with the file system containing the context models or the accessID repository. In case such an error occurs, the developer is asked to retry his previous command. If the error persists, he is asked to forward the error message together with related information to the system manager.

At the moment, error messages are compiled from the information that is available when the error occurs. To improve the quality of the content of error messages, additional mechanisms that traverse the inheritance chain of a context model could be implemented in the future. This would, for example, enable to tell at which level information was defined that is responsible for the new context model to be invalid. Additionally, it could be checked if errors of the same kind occur in the context model before issuing an error report mentioning all of them. This could prevent the need to validate and correct a context model multiple types instead of once.

4.3.5 Minimization

After a context model is considered valid by the validation process, it could be added to the CMR. However, **R2** requires that all context models in the CMR are stored in minimized format. A context model is considered to be in minimal format if it does not specify any information that is already inherited. In order to minimize a context model, the following information needs to be removed:

- Attributes:
  - Type: if multiple types are defined, all types entailed by other types need to be removed. For example, if we define a new node with the type "derived/-boolean, basic/number", "basic/number" needs to be removed from the type specification as it is already contained in the definition of "derived/boolean".

  - Restriction: if a single restriction corresponds exactly with the inherited one, it needs to be removed. For example, we define the restriction "minimumValue = ‘1’, maximumValue = ‘5’" for a node of type number. The inherited restriction is "minimumValue = ‘0’, maximumValue = ‘5’". The restriction "minimumValue" is altered while "maximumValue" is not. Therefore, the minimized restriction is "minimumValue = ‘1’".
  - Reader and writer: The set of access identifiers needs to be removed if it equals the inherited set. As soon as this is not the case, the full set specification needs to be retained. For example, the new context model we defined inherits "reader = ‘a, b, c’". If we defined "reader = ‘b, a, c’ the information can be minimized as the set equals the inherited one. If we defined "reader = ‘a,b’" or reader = ‘a, b, c, d’" the information must not be minimized.

- Value: if the new default value equals to the inherited default value, it needs to be removed.
- Subnodes:
  - Every contained subnode needs to be minimized. If a subnode is inherited, the respective specification in the parent model is relevant. If a subnode is newly defined or specifies additional types, the definition of the referenced types in the CMR is considered.
  - If a minimized subnode does not specify any new or altered information, i.e. no altered attributes, value, or subnodes, it needs to be removed from the context model.

The information can be removed from the context model representation without loosing semantic information. This is because the information is still contained by parent nodes and considered when a context model is resolved or used.

Listing 4.5 shows the fully resolved representation of the context model with the id "minimizationParent" that is considered stored in the CMR.

```
<model type="derived/boolean,basic/composed,basic/number"
           restriction="minimumValue='0',maximumValue='1'"
           reader="a,b,c" writer="a,b">
      1
      <el1 type="basic/composed" reader="*" writer="*" />
      <el2 type="derived/boolean,basic/number"
                 restriction="minimumValue='0',maximumValue='1'"
                 reader="*" writer="*">
            0
      </el2>
      <el3 type="derived/percent,basic/number"
```

```
                   restriction="minimumValue='0',maximumValue='100'"
                   reader="*" writer="*" />
</model>
```

Listing 4.5: Fully resolved representation of the context model "minimizationParent".

Now, we define a new context model, "minimizationChild", that inherits from the type "minimizationParent" as shown in Listing 4.6.

```
<model type="minimizationParent" reader = "a,b" writer = "a,b">
      1
      <el1 type="basic/composed" />
      <el2 restriction="minimumValue='0'" />
      <el3 >
            50
      </el3>
</model>
```

Listing 4.6: Definition of the context model "minimizationChild."

This context model definition is valid. However, many pieces of information that are declared are the same in the type "minimizationParent", such as the set of writer-accessIDs, the value "1", the subnode "el1", or the restriction of the subnode "el2".

The minimized representation, through which all information is contained, is shown in Listing 4.7. This context model can be added to the CMR and complies with R1 and R2.

```
<composed type="minimizationParent" reader="a,b">
  <el3>50</el3>
</composed>
```

Listing 4.7: Minimized Representation of the context model "minimizationChild."

After a context model is validated and minimized, the developer should be presented with the minimized representation. Additionally to this, information about how many and which elements were removed could be displayed. This could lead to the developer better understanding the type system and its capabilities, supporting later work and improving the quality of future context models.

The minimization of context models can be easily conducted by comparing the new context model against the fully resolved representation of its parent model that was already assembled for the validation.

Chapter 5

Implementation

In this section, we will briefly discuss the key elements of the implementation of the CMR. These include the choice of the JDOM framework for the transformation of the data model in Java Objects, the structure of the "Node" class as well as the decision to use SQLite for the AccessID repository.

The logical parts of the CMR are implemented in Java, as is most of DS2OS. This allows for parts of the code, e.g. the validation procedure, to be reused during runtime.

5.1 Data Model Transformation

Parsing a context model from its XML representation to a Java-object structure is a step that is done frequently during resolving and validating a context model. Therefore, this process consists of fetching the stored document, parsing it with an XML-to-Java parser and then transforming it into a Java-object structure fitted to the VSL information components. The implementation of the two latter steps is discussed below.

5.1.1 JDOM Framework

As assessed in Section 4.3.1 it is necessary to transform the XML-representation of context models into Java object structures before an automated mechanism is able to decide if they are valid. In general, two types of parsers exist for parsing XML documents exist: SAX and DOM parsers.

The Simple API for XML (SAX) [33] is an event-based algorithm for parsing XML documents. It linearly parses the documents from top to bottom and creates events for elements, attributes, and values. SAX does not maintain a state and only processes elements one at a time. If it is necessary to keep track of the data the parser has traversed, this information hast to be handled and stored externally. Additionally, using a SAX parser does not allow random access to information. The advantages of SAX are its speed and its low consumption of memory.

The Document Object Model (DOM) [34] is a recommendation of the W3C which defines an interface for programs accessing and modifying information of XML documents. An XML document is represented as a tree structure where every node is an object representing a part of the document. This results in a structured representation of an XML document where information can be freely accessed and modified. However, this approach is slower and requires more memory, as the whole tree structure is stored.

JDOM [35] is a Java-based object model for XML documents. It provides a way for easy and fast document reading and writing. With JDOM, the information of XML files is processed in such a way that we can conveniently retrieve it for our purpose. We can also use JDOM to efficiently construct new context models in the Java routine that can then be saved in an XML file. Constructing context models is required as we store only minimized context models. With JDOM, we do not have to interface a parser directly. JDOM is compatible with both SAX and DOM parsers and combines the flexibility of tree structures with the performance advantages of SAX parsers.

Its object-tree structure makes use of Java collections like List and Array. The root of the tree structure is called "Document". It specifies information about the XML file, such as its URI. A document contains exactly one element, which is the root element. Each element carries information about its value, the name of its tag, and the line it is defined in in the XML document. It also contains a list of its attributes, represented as key-value pairs, and a list of the elements it contains. This rough structure is visualized in Figure 5.1. Note that the nodes contained by "Elements" are references to elements which have the same structure as presented for the root element. Even though more information is contained, this is the only information relevant to us. Because of its performance and convenience, we decided to use the JDOM framework as XML-to-Java parser.

After the XML document was parsed to a JDOM tree, we can be certain that the context model is syntactically valid. To further ensure semantical validity we transform the context model to a Java object fitted to the VSL information model, the class "Node".

5.1.2 Java Class "Node"

As described in Section 4.3.1 we decided that it is necessary to decouple the processing of context models from the JDOM structure. We implemented the Java Class "Node". Each node of the XML data model is represented as an instance of this class. The class consists of variables that are determined by the information describing a context model. An overview of the variable names, types, and purpose is given in Table 5.1. Those are the variables that can be directly determined from the XML file without the need to consult other context models. Additional information is gathered with resolving the

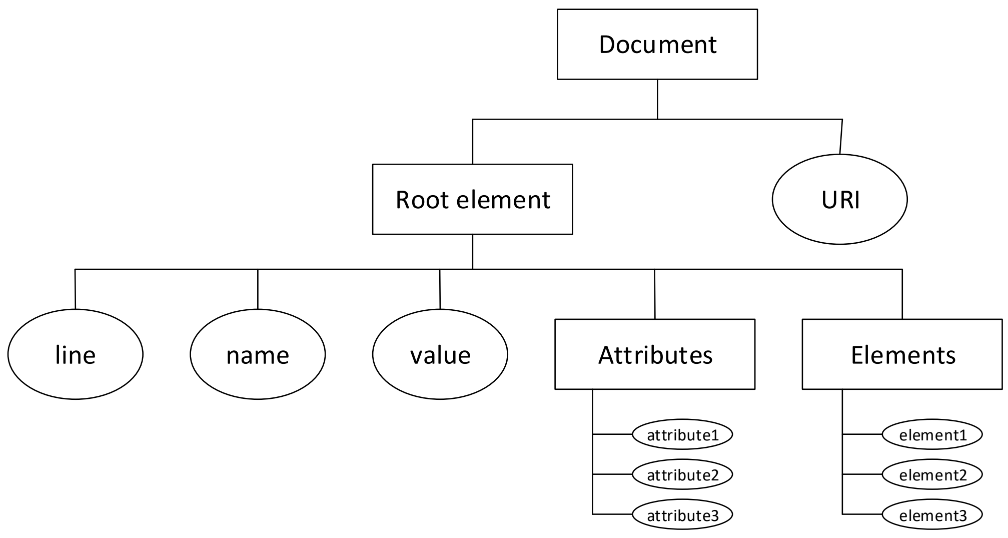


Figure 5.1: Structure of JDOM object trees.

parent, such as the reference to the parent node or the list of validators.

To facilitate later processing this information is transformed further. For example, the class "Node" features a distinct variable for each restriction. In the JDOM tree, all requirements are aggregated under the attribute with the key "restriction". The restriction is parsed by a finite automaton that retrieves the key-value pairs and can detect when the restriction string is malformed.

The VSL information model does not specify any restrictions on the range of the value of numerical values or restrictions. Therefore, we internally represent all information as String. This includes the values of nodes of type "basic/number" and numerical restrictions. The class "Node" provides method for accessing those fields as a BigInteger.

5.2 AccessID Repository

As determined in Section 4.2, accessIDs need to be stored in an independent database. Considering that an accessID is uniquely identifiable by its name and described by additional attributes, such as its description, we decided to use a relational database.

Out of the many existing database engines we decided to use SQLite [36]. SQLite is a lightweight, in-process relational database management system. It is not designed as a client-server architecture, but the database engine is embedded in the application. Calls are directly made to the file holding the data. As no communication to a server is required, this provides a fast access to data. Disadvantages of SQLite are the limited

name	type	description
id	String	The identifier of the containing document.
tag	String	The tag of the node.
line	Integer	The line of the node in the file.
typeString	String	The String defined by the attribute "type".
restrictionString	String	The String defined by the attribute "restriction".
reader	String	The String defined by the attribute "reader".
writer	String	The String defined by the attribute "writer".
value	String	String representing the value of the node. Normalized to omit leading and trailing whitespace.
elements	List<Node>	List containing references to the instances of the node's subnodes.

Table 5.1: Overview of the variables of class "Node" that are directly derived from the XML representation.

size of data (an SQLite database is limited in size to 140 terabytes), the limited number of concurrent writers, and the lack for user management.

Alternatives to SQLite, such as MySQL [37] and PostgreSQL [38], are based on client-server architectures. While they provide more functionality, such as multiple concurrent writers and more distinctive data types, communication is performed through interfaces of sorts, for example ports or sockets. This entails longer response times for requests.

Considering the set-up and requirements for the AccessID repository, it becomes clear that the advantages of SQLite heavily over-weigh its drawbacks. The AccessID repository is not designed to be directly accessed by developers but is rather interfaced through the CMR. The CMR has to refer to the accessID repository for context model validation. This makes fast accessibility of data very important. As the accessID repository is only accessed through the CMR, concurrent write operations are not required. Access control can be managed on by the CMR. The size limit imposed by SQLite does not constitute a problem, as the size of the data stored in the CMR is rather small.

Currently the accessID repository is designed to contain one table. A tuple consists of the attributes "accessID", "description", and "submitted on". "accessID" is the primary key.

If a context model defines new accessIDs, the validation process queries the accessID repository for the existence of the respective accessIDs. Retrieving and submitting new accessIDs to the accessID repository can be conducted through the CMR.

Chapter 6

Evaluation

After implementing the CMR, we will now evaluate our implementation for correctness, performance, and usability in the following sections.

6.1 Correctness

One of the important properties of the CMR is that **R1** and **R2** are fulfilled correctly. Validation and minimization performed by the CMR must be correct by all means.

The correctness of the validation and minimization process is of high importance due to two reasons:

1. It is important that only valid context models are used by orchestration services. Using a non-valid context model could lead to unexpected behavior of the system.
2. Our processing is based on the assumption that all context models in the CMR are valid and minimized. When faulty processing accepts an invalid context model the knowledge base of the CMR is corrupted. As our assumption is now no longer fulfilled, validation and minimization may fail completely.

As we cannot formally verify the correctness of the validation process, we did extensive testing of the implementation. By default, the CMR accepts a context model unless it detects an invalid declaration. Therefore, we created a set of context models that are invalid and need to be rejected. These context models cover all aspects of the invariants described in Section 2.4.5 and the corresponding validators enumerated in Section 4.3.2.

We created 70 invalid models to evaluate the validation process together with 19 valid models used as base to model invariants affected by inheritance. These models were used throughout the implementation process to validate the functionality and to detect when invalid models were still accepted. All invalid models are rejected with the expected error message by the final implementation.

Models are only minimized when they are valid. We created a second set of context models that require minimization in all ways, i.e. minimization of the restrictions, the accessID, the value, and the subnodes.

For both sets, files containing the expected results – either the error message or the minimized representation – are available. Those results were created by running the current implementation and then manually verified to be as expected. By using the results, the correctness of the processes can also be automatically ensured after future modifications.

6.2 Performance

In this section, we will evaluate how resolving, validation, and minimization perform in regard to different context model properties.

6.2.1 Creation of Test Models

To interfere the general tendency of how the runtime of our processes corresponds with context model structure and size, we will consider two variables: (1) Inheritance depth. This variable denotes how many types a context model inherits from in a linear inheritance chain. (2) Model complexity. This variable describes the total number of nodes a context model contains.

We will consider context models only inheriting from non-composed structures, context models with a high number of subnodes, as well as context models with a mix of inheritance depth and subnodes.

Currently only a very small set of context models is available that is not sufficient for drawing conclusions about the performance of the CMR for larger and more complex context models. Therefore, we created generic context model structures with the desired features. We present the properties of the context models we created below. Afterwards, we summarize the important results of our tests.

6.2.1.1 Inheritance Depth

To evaluate the impact of the inheritance depth on the processing time, we created three different inheritance chains consisting of 200 context models each.

The basic type for the context models in the first inheritance chain is "basic/number". The root node of the context model "typeNumber0" inherits from it. The context model called "typeNumber[i]" inherits from the type "typeNumber[i-1], where *i* ranges from 0 to 199. None of the context models in the chain defines a value, restriction, or accessID.

This set of models is intended for determining the sole influence of the inheritance depth without the influence of possible side effects from further specifications. The context model "typeNumber43" is shown exemplaryily in Listing 6.1. We will refer to this set of context models as set 1.

```
<model43 type=".../set1/typeNumber42" />
```

Listing 6.1: Definition of "typeNumber43" in set 1.

We used the second set of context models to determine whether the impact of inheritance depth is similar to the first one. It also consists of a inheritance chain of "basic/number". The difference is that the context models in this set specify values, restrictions, and accessID which change for every context model. They are determined by the inheritance depth of each context model. Information contained by the context models in the inheritance chain is considered unnecessary, as it is overwritten by the last context model. By comparing the performance of set 1 to set 2, we want to determine the overhead of resolving such unnecessary information. An example of a model out of this set is shown in Listing 6.2.

```
<model53 type=".../set2/typeNumber52" reader="b" writer="b"
            restriction="minimumValue='53',maximumValue='1947'">
      53
</model53>
```

Listing 6.2: Definition of "typeNumber53" in set 2.

The third set of context models is based on "basic/text" instead of "basic/number". Throughout the inheritance chain, the restriction, value and accessIDs alternate. We examine this set of context models because the restriction "regularExpression" of type "basic/text" is not overwritten, but extended through inheritance. We expect that this entails that more work has to be put in the validation of this restriction. Therefore, the validation process should take longer. An example context model of this set is shown in Listing 6.3.

```
<model75 type=".../set3/typeText74" reader="b" writer="b"
            restriction="regularExpression='b*a*'">
      b
</model75>
```

Listing 6.3: Definition of "typeText75" in set 3.

6.2.1.2 Number of Subnodes

In order to evaluate the impact of the number of subnodes on the process time, we use context models of type "basic/composed" that contain a tree structure of nodes. This tree structure is determined by its degree, or branching factor, and its depth. A context model of depth 0 only contains the root node, one of depth 1 contains the root node and *degree* direct subnodes. In a model of depth 2 theses subnodes contain in turn again *degree* subnodes.

The total number of nodes such a context model contains can be calculated by

$$\sum_{i=0}^{depth} degree^{i}.$$

We do use two different kinds of such context models. The subnodes in context models of the first kind are all of the same type "base". We refer to this kind of context model as "context model with uniform subnodes". An example for this kind of context model is given in Listing 6.4.

```
<model type="basic/composed">
  <el0 type=".../base" />
  <el1 type=".../base" />
</model>
```

Listing 6.4: Definition of a context model with uniform subnodes, called "type_degree2_depth2".

The second kind of context models contains subnodes which are all of different types. The i^{th} subnode is of type "base[i]". A context model of this kind which contains n subnodes refers to n different types. We refer to this kind as "context model with different subnodes". An example for this kind of context model is given in Listing 6.5.

```
<model type="basic/composed">
  <el0 type=".../base1">
    <el0 type=".../base2" />
    <el1 type=".../base3" />
  </el0>
  <el1 type=".../base4">
    <el0 type=".../base5" />
    <el1 type=".../base6" />
  </el1>
</model>
```

Listing 6.5: Definition of a context model with different subnodes, called "type_degree2_depth2".

For both types, we will consider context models with degree four and depth two, which equals a total number of 21 nodes, context models with degree four and depth four, which equal a total number of 341 nodes, and context models with degree four and depth six, which equals a total number of 5461 nodes.

6.2.1.3 Mixed Structure

In reality, context models will have a mixed structure that includes both a certain inheritance depth and the containment of subnodes. Based on our current experience, we expect the majority of context models to have an inheritance depth of 5 or lower and to contain at most 20-40 subnodes. Based on these estimations, we created a set of context models based on "basic/composed". With every inheritance step, eight new subnodes are specified. So a node on inheritance depth i defines 8 new nodes and additionally inherits $8i$ nodes.

6.2.2 Execution Time

Considering the design of the processes, we expect the general behavior to be as following:

- **Resolving** only depends on the number and complexity of context models in the inheritance chain. Therefore, the more context models are inherited from and the more complex they are, the longer resolving should take. The time required for resolving should be independent from the new context model.
- **Validation** does not depend on all context models in the inheritance chain, only on the new context model and the direct parent model. The current algorithm traverses the new context model and compares it to the parent. Therefore, the runtime of validation should only depend on the complexity of the new context model. The complexity of the new context model is primarily determined by the number of subnodes it contains, as they all need to be validated independently. Factors like how many restrictions, reader, or writer attributes are specified or how many and what types the nodes inherit from also play a smaller role.
- **Minimization** is structured just as validation. Therefore, the statement made about validation also fits to minimization. However, minimization should take shorter than validation of the same context model. Minimization consists merely of comparing the existence of information, while validation requires more complex logic.

For the context models described with the above mentioned variables this entails:

- A relatively small context model, i.e. one that does not contain subnodes, with a very high inheritance level will require time proportional to its inheritance depth for resolving. This is because all context models in the inheritance chain need to be traversed. However, the time for validation and minimization should be constant, as this process compares the new context model only against its parent model.

- A complex context model, i.e. one the contains many subnodes, with a low inheritance hierarchy will require time proportional for validation and minimization. However, as only few nodes need to be traversed for resolving, this time should be constant when the traversed context models do not introduce much complexity.

6.2.2.1 Impact of the Inheritance Depth

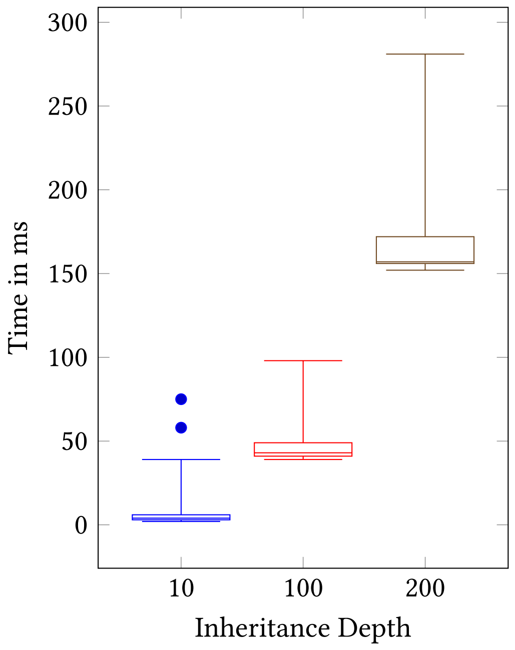
To estimate the impact of the inheritance depth, we conducted the following tests:

- Set 1: we performed the resolving process for the context models "typeNumber9", "typeNumber99", and "typeNumber199" 200 times. We measured he time for each run separately. As validation and minimization are much faster for these sets, we run each of these processes 1000 times and measure the combined duration.
- Set 2: we perform the same test as above.
- Set 3: we perform the same test as above for the context models "typeText9", "typeText99", and "typeText199".

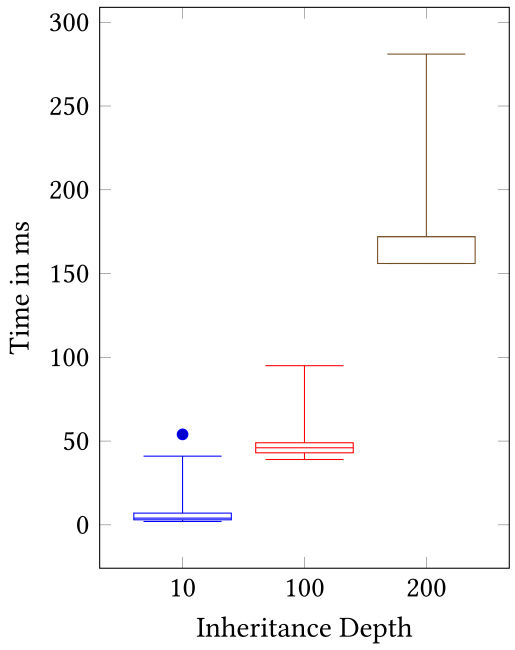
We expect the following results:

1. For all sets, we expect an increase in the time required to perform the resolving with an increased inheritance depth.
2. For sets 1 and 2, we expect similar resolving time for the same inheritance depth. We expect set 2 to require slightly more time as new values are defined in every step.
3. For set 1 and set 2, we expect the time required for validation to remain constant.
4. For set 3, we expect a slightly increase in validation time with increased inheritance depth. We expect this as the restriction "regularExpression" is not overwritten, but appended with the inherited ones.
5. For all sets, we expect the time required for minimization to remain constant.

Figure 6.1 compares the runtime of the resolving process for the types "typeNumber9, "typeNumber99", and "typeNumber999" for both set 1 and 2. As we can see, the resolving time increases with the inheritance depth as expected (1). The result for set 3 is similar. Set 2 only takes negligibly longer than set 1, other than assumed (2). This leads us to the conclusion that it has no influence on the performance of the resolving process whether new values or attributes are specified or not.



(a) Resolving time for context models of set 1.



(b) Resolving time for context models of set 2.

Figure 6.1: Time in milliseconds required for resolving context models with inheritance depth 10, 100, and 200 for set 1 (a) and set 2 (b).

	depth 10	depth 100	depth 200
set1/typeNumber	4 ms	3 ms	2 ms
set2/typeNumber	712 ms	804 ms	815 ms
set3/typeText	824 ms	877 ms	1180 ms

Table 6.1: Time in milliseconds for performing only validation 1000 times for the inheritance depths 10, 100, and 200. The time required for resolving is not included.

The validation time for set 1 and set 2 was indeed constant for different inheritance depths (3), as shown in Table 6.1. For set 3 the same did apply. This objects our assumption (4). It is observable that the time required for validation is higher when the context model contains more information: validation of set 1 takes significantly shorter than validation for set 2 and 3.

The minimization process took equally long for all modes with around 30 milliseconds required to perform minimization 1000 times (5).

6.2.2.2 Impact of the Number of Subnodes

To estimate the impact of the number of subnodes, we conducted the following tests:

- We performed the resolving, validation, and minimization process for the context models "type_degree4_depth2", "type_degree4_depth4", and "type_degree4_depth6" 200 times. This was conducted for both context model types. This corresponds to the total numbers of nodes of 21, 341, and 5461. For each run, we measured the time for each of the processes separately.

- We performed the same tests as above, but disabled the caching component for resolved context models.

We expect the following results:

1. For all context models, we expect the average resolving time with and without cache to be constant, equal, and negligibly small. We assume this because the only context model being resolved is the context model "basic/composed".
2. For both context model types, we assume that the validation time increases significantly with the number of nodes when the cache is disabled. This is because the subnodes of the new context model need to be resolved during validation, as they are not defined in the parent model.
3. For the context model with uniform subnodes, we expect the validation time to be significantly less when caching is enabled. Not as many accesses to the file system and parsing steps are required.
4. For the context models with different types of subnodes, the validation time will be less when caching is enabled, but still more than for the context model with uniform subnodes.
5. Minimization time should increase with the number of subnodes and should not depend on caching.

The results for the resolving process are as expected. In most cases, the resolving was so fast that it could not be recorded with milliseconds and are therefore listed as zero. For all context models, the highest outliers for validation was 32 milliseconds. This corresponds to our assumption (1).

Figure 6.2 depicts the results for validation conducted without caching. As we presumed (2), the runtime significantly increases with the number of nodes. What we did not anticipate is that the context models with different subnodes tend to require less time than the models with uniform subnodes of the same size. We assume this is due to the different types we used for the base types of the subnodes. For the context model with different types, all base types inherited from only "basic/composed". For the context model with uniform subnodes, the base type inherits from "basic/composed" and "basic/number". As those two basic context models need to be retrieved for the validation of every subnode, more runtime is required.

The test further supports our assumption that the validation time significantly declines when caching is enabled (3). The validation process was unexpectedly even so fast that

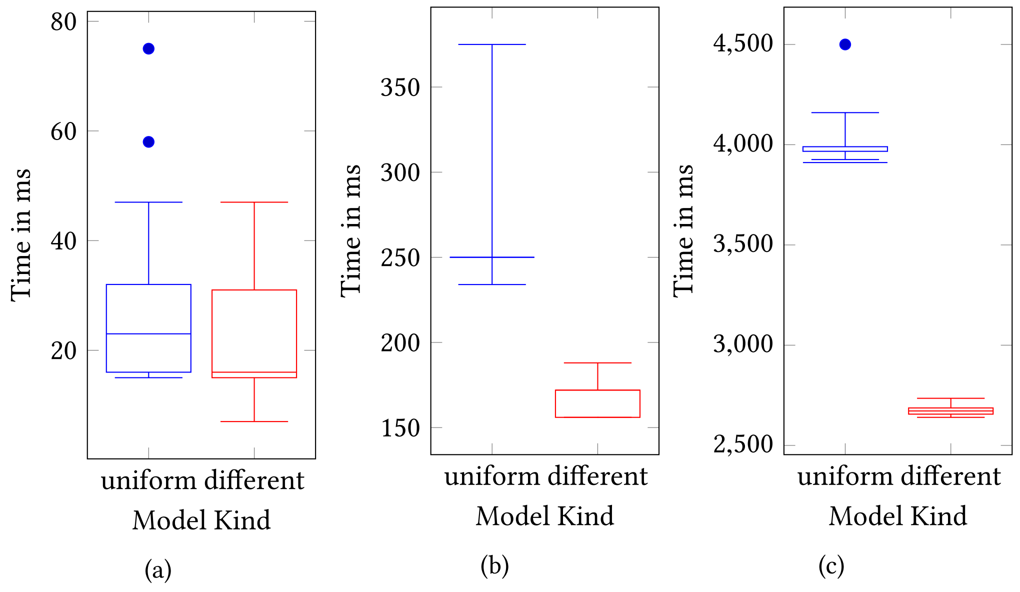


Figure 6.2: Time in milliseconds required for validating context models with 21 (a), 341 (b), and 5461 (c) nodes. Blue depicts the context model with uniform types, red depicts the context model with different nodes. Note the different scales on the y axis.

we were not able to measure the time by the means of milliseconds.

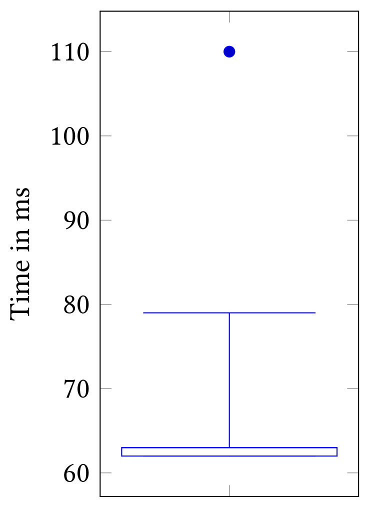
With the results depicted in Figure 6.3 we can support our assumption (4). The validation time is roughly half of the one required when caching is disabled. Even though the types of the of the subtypes cannot be cached, the basic type type inherit from is cached. This reduces the number of necessary accesses to context models stored in plain text files.

The fifth assumption is also fostered by our results. Even though the time required for the context models with 21 and 341 was so short that we were not able to measure them exactly, the context models with 5461 nodes behaved just as expected. Figure 6.4 shows that the minimization time is constant over context model type and over enabling and disabling caching.

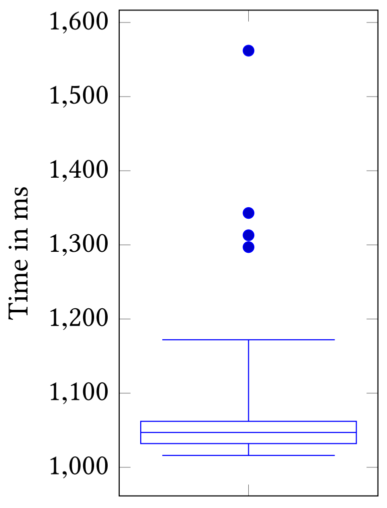
6.2.2.3 Mixed Models

To show that the two context model structures discussed above do have the same impact when mixed, we performed test on context models with mixed structures.

The results were as expected: the more complex the context model we inherit from, the longer resolving takes. The more subnodes a context model contains, the longer validation and minimization take. When caching is enabled, the validation time heavily depends on the variety of subnode types.



(a)



(b)

Figure 6.3: Time in milliseconds required for validating the context model "type_degree4_depth6" with uniform nodes (a) and with different nodes (b). Note the different scales on the y axis.

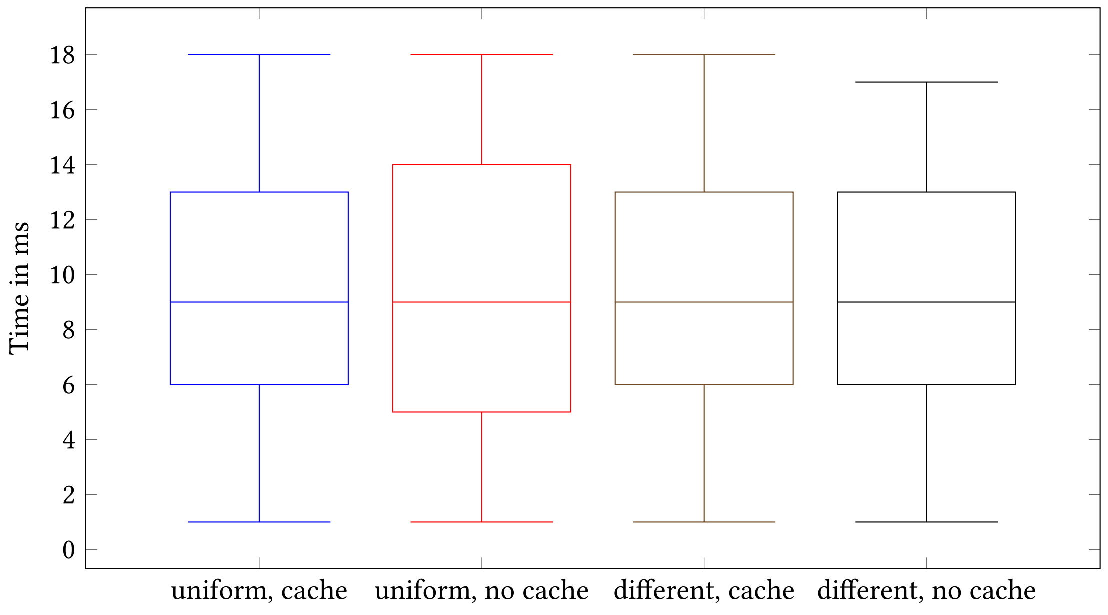


Figure 6.4: Time in milliseconds required for minimizing context models with 5461 nodes. Variation on the type of subnodes and on enabling and disabling caching.

6.2.3 Summary

We have shown that context models much more complex and with a higher inheritance depth than we anticipate to be submitted to the CMR in the future still only require processing time that comfortably fits in our aspired response time that allows interactive responses (**R6**).

Our evaluation showed that the processing time heavily depends on the number of context models that need to be retrieved and parsed. We have already dampened this effect by using a cache storing all context models retrieved in the same process. Additionally, the time required for parsing should be further reduced.

Rating the performance achieved as satisfying relies on the assumption that context models that are submitted to the CMR are created by humans. Consequently, the complexity of context models should lie within the boundaries tested above. However, context models could be automatically generated to be complex and demanding on the resources of the CMR.

6.3 Usability

With **R8** we demand that the error reports returned for invalid context models are helpful in finding, understanding, and resolving the error. To assess the quality of the currently used error reports, a survey with 10 participants was conducted.

At the beginning, the participants had to state their level of knowledge of the VSL information model. Four participants stated to have no previous knowledge, two basic knowledge and four good knowledge. Afterwards, a short introduction to the VSL information model was given.

The participants were presented five different invalid context models together with the error report returned by the CMR. Participants were asked why the context model was invalid or how they could resolve the error. Then they rated how helpful the error report was in finding and resolving the error. The options were "not helpful", "helpful", and "very helpful". No participant used the rating "not helpful" at any time.

We will present the results for the different error reports below. We give the percentage of participants that were able to give the correct answer and consider their ratings for the helpfulness of the error report. Table 6.2 gives an overview of the results.

The first invalid context model's root node contained a subnode even though it was neither of type "basic/list" nor type "basic/composed". 80 percent of the participants were able to resolve the error, the rest falsely wanted to change the type definition of the subnode instead of the root node. Out of those answering correctly, seven found the message to be "very helpful". Half of the answers considered the message "very helpful"

		error finding			error resolving		
error type	#of correct answers	+	o	-	+	o	-
containing subnode	8	7	1	0	4	4	0
relaxing restriction	9	7	2	0	7	2	0
wrong restriction	9	9	0	0	6	3	0
undefined accessID	10	9	1	0	5	5	0
multiple	10	9	1	0	9	1	0

Table 6.2: Number of the correct answers for each context model, together with the rating of those participants who answered correctly.

for resolving the error. As it seams that understanding the error was easy, but resolving was not as much supported by the error report, additional information might be added. "Maybe you can add "basic/composed" to the type of [faulty node]" could be one way to do this.

The second invalid context model specified the restriction "maximumValue=2", while the inherited restriction was "minimumValue=1". 9 of the participants were able to tell why the context model was invalid. Out of those, 7 found the message "very helpful" for finding and resolving the error.

The third invalid context model was of type "basic/text" and specified the restrictions "minimumValue" and "maximumValue". Nine out of the participants were able to resolve the error by making the context model of type "basic/number" or using "regularExpression" instead of the afore mentioned restrictions. All of those found the error message "very helpful" for finding the error, while only six found it "very helpful" for resolving the error. We consider this as a hint that more guidance for resolving the error is desired in this case.

The fourth context model used an accessID not defined in the accessID repository. All participants were able to resolve this error by either adding the accessID to the repository or by using an already defined one. Nine participants found the message to be "very helpful" for finding errors. Only half of the participants found the message to be "very helpful" to resolve. For this kind of error further guidance may be necessary.

The last context model was invalid at several spots. The root node did specify a restriction not defined for its type. The subnodes' values were invalid, one due to not complying with the restrictions and once due to not complying with the type. To evaluate whether developers found the display of multiple error messages supportive, we assembled the error message in Listing 6.6 from the separate ones, as this feature is not implemented yet.

```
Error at node ’myComposed2’ (line 1):
This node may not define restrictions for the minimum and maximum
of a value as the node is not of type ’basic/number’.

Error at node ’myComposed2/el1’ (line 2):
The value ’3’ is too large.
Restriction ’maximumValue’: 1.

Error at node ’myComposed2/el3’ (line 8):
The value ’three’ cannot be parsed as a number.
The node inherits from type ’basic/number’, therefore its value has
to be a number.
```

Listing 6.6: Error-report detailing multiple errors at once.

The feedback to this error report was positive: All participants were able to recognize the errors. Nine found found the error message "very helpful" for finding and resolving the error. This encourages us to use such error messages, which require to go on with the validation of the next subnodes even after one error was already found.

We interpret these results that the error reports produced by the CMR are indeed helpful for finding and understanding the error and that **R8** is fulfilled. However, more needs to be done to support the resolving of errors. This is also underlined by the textual feedback given by some participants. They suggested to give more details about the error. In case of inheritance errors, for example, error messages could be improved by displaying the parent model. In case of restriction errors it was suggested to display the rules for restrictions.

6.4 Fulfillment of Requirements

The implementation that we developed during this work covers the core processes of the CMR: validation (**R1**) and minimization (**R2**). In the evaluation we showed that the implementation complies with **R6** and processes context models efficiently. It is possible to submit context models to the CMR (**R4**). Context models can also be retrieved, but knowledge of its existence and identifier are required, as there is no possibility yet to browse or search context models. We can draw the conclusion that the error reports are helpful for finding errors from the survey we conducted (**R8**). However, participants expressed the wish to be supported more with resolving an error.

Table 6.3 states which of the requirements we identified are fulfilled by our implementation.

The requirements **R3**, **R5**, **R7** were not dealt with in this work. **R3** and **R7**, which

Requirement		
R1: Validation	++	implemented and tested
R2: Minimization	++	implemented and tested
R3: Browsing and Searching	-	not implemented
R4: Submission and Retrieval	+	submission possible, retrieval when existence of model is known
R5: Convergence Mechanisms	-	not implemented
R6: Fast Validation Process	++	
R7: Intuitive Browsing	-	not implemented
R8: Helpful Error Reports	+	basic, only one error at a time, more support possible

Table 6.3: Fulfillment of the requirements identified in Section 2.6.

concern the browsing and searching functionality of the CMR, can be fulfilled by implementing a web application for the CMR where developers can browse and submit context models. **R5** requires standardization of models. First approaches for this were presented in this work.

Chapter 7

Conclusion

With this work, we laid the foundation for the crowdsourced creation of context models for DS2OS.

We illustrated why the introduction of a component like the CMR satisfies the desired properties of a crowdsourcing framework for DS2OS. By considering the role of the CMR in the DS2OS and anticipating how it will be used, we identified the detailed requirements for the CMR. We also considered problems that arise with collaborative crowdsourcing. Context models may not be created by using already existing functionality or may no be suitable to be reused themselves. This would hinder standardization of interfaces.

We compared the capabilities of the proposed CMR with existing solution approaches for similar applications. We argued that these existing solutions do not fulfill all the expectations we have for the crowdsourcing framework for the DS2OS. However, some of their approaches could be an enrichment for our system, such as semantic tags for context models and providing adapted editors.

Ensuring that only valid context models may be submitted is one of the main responsibilities of the CMR. As standard validation mechanisms for XML cannot validate the semantic component of the VSL information model, a custom solution needed to be implemented. The same applies for the minimization of context. By making this implementation independent from the currently used data model, XML, adaptations of additional data models are possible in the future.

Our design splits the functionality into different modules: resolving, validation, and minimization. Those modules can be reused for different purposes. Resolving can be reused for retrieving a resolved model from the CMR, either for displaying it to a developer or for deploying it in a smart space. The validation mechanism also can be reused on deployment.

The evaluation showed that our implementation achieves our goal: the requirements most important for the basic functionality of the CMR are fulfilled. We are able to validate and minimize models automatically and with good performance. The error reports proved as being helpful in understanding errors.

7.1 Future Work

Future work should first focus on approaching the requirements not considered by our implementation yet. To make context model reuse possible, a graphical interface where the CMR can be browsed and searched for its context models is necessary.

Approaches for converging mechanisms leading to standardization of context models were presented in Section 2.5.4. Those should be considered more closely and be implemented. Our survey showed that the error reports could be improved to support resolving an error. Mechanisms trying to guess the developer's intention and suggesting corrections are an option as well as displaying several error messages at once.

To facilitate searching context models in the CMR, a tagging system could be introduced as mentioned in Section 3.1. For storing this meta-information, a database decoupled from the storage of the context models would be necessary.

Malicious intentions of context model contributers must be considered. Our implementation of validation and minimization performs well. But automatically generated context models that are artificially complex have the power to consume the resources of the CMR and prevent it from replying to other requests. Security mechanisms that are able to detect such attacks should be deployed.

Besides the CMR, an other component is important for fostering the crowdsourced creation of context models. Providing an adapted editor that is tailored to the VSL context model relieves the manual work required for the creation of context models. Such an editor could feature, for example, syntax-highlighting, auto-completion, and in-editor validation.

Bibliography

[1] M.-O. Pahl, G. Carle, and G. Klinker, "Distributed smart space orchestration," in *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, 2016, pp. 979–984.

[2] M. Weiser, "The computer for the 21st century," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 3, no. 3, pp. 3–11, 1999.

[3] M.-O. Pahl, "Distributed smart space orchestration," Doctoral Dissertation, Technische Universität München, München, 2014.

[4] M.-O. Pahl and G. Carle, "Crowdsourced context-modeling as key to future smart spaces," in *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, 2014, pp. 1–8.

[5] R. Minerva, A. Biru, and D. Rotondi, "Towards a definition of the internet of things (iot)," *IEEE Internet Initiative*, vol. 1, 2015.

[6] H. J. Kim, I. Kim, and H. G. Lee, "The success factors for app store-like platform businesses from the perspective of third-party developers: An empirical study based on a dual model framework," in *Proceedings of the Pacific Asia Conference on Information Systems 2010*, 2010, p. 60.

[7] D. Hovland, "The inclusion problem for regular expressions," *Journal of Computer and System Sciences*, vol. 78, no. 6, pp. 1795–1813, 2012.

[8] World Wide Web Consortium, "Extensible markup language (xml)," 13.01.2018. [Online]. Available: https://www.w3.org/XML/

[9] ISO and IEC, "Iso/iec directives, part 1," 2012.

[10] "Extensible markup language (xml) 1.0 (fifth edition)," 18.02.2018. [Online]. Available: https://www.w3.org/TR/xml/

[11] World Wide Web Consortium, "Xml schema," 13.01.2018. [Online]. Available: https://www.w3.org/XML/Schema

[12] David Gaßmann, "Implementation of a context model repository," Bachelor's Thesis, Technische Universität München, München, 2015.

[13] T. R. Gruber, “A translation approach to portable ontology specifications,” *Knowledge acquisition*, vol. 5, no. 2, pp. 199–220, 1993.

[14] L. Stojanovic, “Methods and tools for ontology evolution,” 2004.

[15] World Wide Web Consortium, “Semantic web,” 04.02.2018. [Online]. Available: https://www.w3.org/standards/semanticweb/

[16] ——, “W3c semantic web activity,” 04.02.2018. [Online]. Available: https://www.w3.org/2001/sw/

[17] ——, “Rdf - semantic web standards,” 03.02.2018. [Online]. Available: https://www.w3.org/RDF/

[18] “Rdf 1.1 turtle,” 23.01.2018. [Online]. Available: https://www.w3.org/TR/turtle/

[19] “Rdf 1.1 json alternate serialization (rdf/json),” 02.10.2017. [Online]. Available: https://www.w3.org/TR/rdf-json/

[20] “Rdf 1.1 xml syntax,” 02.10.2017. [Online]. Available: https://www.w3.org/TR/rdf-syntax-grammar/

[21] “Owl - semantic web standards,” 18.02.2018. [Online]. Available: https://www.w3.org/OWL/

[22] T. Tudorache, N. F. Noy, S. Tu, and M. A. Musen, “Supporting collaborative ontology development in protégé,” in *International Semantic Web Conference*, 2008, pp. 17–32.

[23] C. W. Holsapple and K. D. Joshi, “A collaborative approach to ontology design,” *Communications of the ACM*, vol. 45, no. 2, pp. 42–47, 2002.

[24] S. Karapiperis and D. Apostolou, “Consensus building in collaborative ontology engineering processes,” *Journal of Universal Knowledge Management*, vol. 1, no. 3, pp. 199–216, 2006.

[25] T. Reschenhofer, M. Bhat, A. Hernandez-Mendez, and F. Matthes, “Lessons learned in aligning data and model evolution in collaborative information systems,” in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 132–141.

[26] “Project haystack.” [Online]. Available: https://project-haystack.org/

[27] V. Charpenay, S. Käbisch, D. Anicic, and H. Kosch, “An ontology design pattern for iot device tagging systems,” in *5th International Conference 2015*, 2015, pp. 138–145.

[28] “The cellml project.” [Online]. Available: https://www.cellml.org/

[29] World Wide Web Consortium, “W3c math home,” 28.01.2018. [Online]. Available: https://www.w3.org/Math/

[30] C. M. Lloyd, J. R. Lawson, P. J. Hunter, and P. F. Nielsen, “The cellml model repository,” *Bioinformatics (Oxford, England)*, vol. 24, no. 18, pp. 2122–2123, 2008.

[31] J. Nielsen, *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, 1994.

[32] B. Shneiderman, “Designing computer system messages,” *Commun. ACM*, vol. 25, no. 9, pp. 610–611, 1982.

[33] “Simple api for xml (sax).” [Online]. Available: http://www.saxproject.org/

[34] “Dom parsing and serialization,” 22.02.2018. [Online]. Available: https://www.w3.org/TR/DOM-Parsing/

[35] “Jdom,” 22.02.2018. [Online]. Available: http://www.jdom.org/

[36] “Sqlite,” 22.02.2018. [Online]. Available: https://sqlite.org/index.html

[37] “Mysql,” 22.02.2018. [Online]. Available: https://www.mysql.com/de/

[38] “Postgresql: The world’s most advanced open source database,” 22.02.2018. [Online]. Available: https://www.postgresql.org/