OPC UA Address Space Transformations

Tomi Tuovinen

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 01.03.2016

Thesis supervisor and advisor:

D.Sc. Ilkka Seilonen

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Tomi Tuovinen

Title: OPC UA Address Space Transformations

Date: 01.03.2016 Language: English Number of pages: 7+43

Department of Electrical Engineering and Automation

Professorship: Information and Computer Systems in Automation

Supervisor and advisor: D.Sc. Ilkka Seilonen

OPC UA has begun to take over its predecessor's place in the development of machine-to-machine communications and data transfer. OPC as a technology is one of the more popular choises when it comes to communications with field devices, regardless of the developer of the device itself. To further improve the features available to OPC UA developers, the spread and usage of the technology can be improved even further. One such feature that could be seen as potentially powerful in utilizing the core features and design ideas of OPC UA is address space transformation.

By designing a model transformation language for OPC UA address spaces, the technology will be given another advantage compared to OPC classic. The transformation language would allow OPC UA to utilize its server-oriented architecture by enabling more simplified server aggregation, and customization of the displayed address space on the aggregating server's side. Since there has not been much public research in such transformation languages for OPC UA address spaces, the first step taken in this thesis is to discover the requirements for such a language based on other modeling technologies and the transformations used in those. The second step is to apply that knowledge to design a functional rule language and a transformation engine to perform the necessary transformations.

The designed rule language and transformation engine are capable of performing the transformation required by the test cases described in this thesis. From the results of those cases, the language shows that it has the necessary features to perform at least the most common types of transformations. Further development is required for more advanced transformations and uncommon scenarios that did not appear in the test cases. The information gathered in both the requirements and the design parts of this work will regardless be useful in the future development of a transformation process for OPC UA address spaces.

Keywords: OPC UA, Address space, Transformation, Aggregation

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Tomi Tuovinen

Työn nimi: OPC UA:n nimiavaruuksien muunnokset

Päivämäärä: 01.03.2016 Kieli: Englanti Sivumäärä: 7+43

Sähkötekniikan ja automaation laitos

Professuuri: Automaation tietotekniikka

Työn valvoja: TkT Ilkka Seilonen

Työn ohjaaja: TkT Ilkka Seilonen

OPC UA on alkanut ottaa edeltäjänsä paikkaa laitteiden välisessä kommunikoinnissa ja datansiirrossa. OPC on ollut suosituimpien teknologioiden kärkisijoilla jo pitkään, kun on ollut tarpeen yhdistää eri valmistajien kenttälaitteita samaan verkkoon tiedonjakoa varten. OPC UA:n leviämistä laajempaan käyttöön voidaan parantaa lisäämällä mahdollisten ominaisuuksien määrää entisestään. Yksi mahdollinen ominaisuus, joka pystyy hyödyntämään OPC UA:n perusominaisuuksia tehokkaasti, onnimiavaruuksien muunnokset.

Suunnitelemalla address spacejen muunnoksia varten kielen, OPC UA saa uuden edun klassiseen OPC:hen verrattuna. Muunnoskieli yhdistettynä OPC UA:n palvelinarkkitehtuuriin mahdollistaisi yksinkertaisen aggregoinnin, jolla ulospäin näkyvien nimiavaruuksien ulkoasua voitaisiin muokata tehokkaasti. Koska tähän aiheeseen liittyen ei ole tehty paljoa julkista tutkimusta, täytyy työssä ensin selvittää, mitkä ovat tällaisen muunnoskielen vaatimukset käyttäen hyödyksi vastaavanlaisia teknologioita ja muunnoskieliä, joita on käytetty muissa yhteyksissä. Tätä tietoa hyödyntäen voidaan suunnitella ja toteuttaa sääntökieli ja kieltä lukeva työkalu, jolla halutut muunnokset voidaan suorittaa.

Suunniteltu sääntökieli ja työkalu onnistuivat muuntamaan halutut nimiavaruudet, jotka oli määritelty tässä työssä esitellyissä testitapauksissa. Testien lopputuloksista voidaan päätellä, että suunniteltu kieli sisältää tarpeeksi ominaisuuksia, jotta se pystyy toteuttamaan kaikki perustapaukset muunnoksissa. Lisätutkimus on kuitenkin tarpeen, jotta kehittyneemmät tapaukset ja erikoistilanteet saadaan käsiteltyä oikein tilanteissa, joita ei pystytty kattamaan testitilanteilla. Kielen vaatimusten määrittely ja suunnittelu kuitenkin antavat tarpeeksi tietoa tällaisten sääntökielien toteuttamisesta, jotta tarpeellista jatkokehitystä voidaan alkaa toteuttaa.

Avainsanat: OPC UA, nimiavaruus, muunnos, aggregointi

Preface

I want to thank my supervisor D.Sc. Ilkka Seilonen for providing me a very intersting and unique thesis subject to work on, and the multiple companies experssing interest in the progress of my work for showing me that this research holds real value in the eyes of the developers of OPC UA and related products.

Espoo, 08.03.2016

Tomi P.-P. Tuovinen

Contents

Symbols and abbreviations

Abbreviations

OPC	Open Platform Communications
OPC UA	OPC Unified Architecture
SCADA	Supervisory Control And Data Acquisition
HMI	Human-Machine Interface
XML	eXtensible Markup Language
XSLT	Extensible Stylesheet Language Transformations
COM	Component Object Model
DCOM	Distributed Component Object Model
MDE	Model-Driven Engineering
QVT	Query/View/Transformation
PDF	Portable Document Format
RDF	Resource Description Framework
OWL	Web Ontology Language
SQL	Structured Query Language
DSL	Domain Specific Language
LHS	Left Hand Side
RHS	Right Hand Side
BNF	Backus-Naur Form
ABNF	Augmented Backus-Naur Form
EBNF	Extended Backus-Naur Form
HDA	Historical Data Access

1 Introduction

1.1 Background

In automation industry, field devices are key to solving most of the tasks in the field, and the devices come from multiple different vendors, each having their own way of performing communications. OPC was originally designed as a common specification for field device communications and data transfer, and OPC UA is essentially version 2.0 of that original idea, but expanded to be used for much more than just field devices.

The software development industry is currently very interested in development on Internet of Things, Big data, and cloud-based software solutions. While OPC UA is traditionally known for its function as a communication framework in automation processes such as manufacturing, it can be easily adapted to be used in other software domains. While it remains yet to be seen how well OPC UA will spread to these functions, developing features such as data aggregation for it will allow a much wider range of design to be created using it. While OPC UA itself does not directly provide any specific support to being used as a server platform in the cloud, there are no restricting factors for it either. Being able to transform address space models and create data connections directly in the aggregating server itself to provide an interface of combined data nodes would give OPC UA an edge over other data transfer technologies in this field.

There has not been much research done on OPC UA server aggregation. Aggregation in this sense means the connection of multiple servers into one, combined server, which provides an interface for accessing the exposed data of all underlying servers. Grosmann designed a functioning OPC UA aggregation server, which was capable of connecting most OPC UA features, such as read and write calls, to the underlying servers [18]. Elovaara expanded on the designed aggregate server by implementing a preliminary transformation language that was capable of adjusting the address space being displayed at the aggregate server's side through the creation of a set of rules that dictatet how the new address space should look [14]. While it did perform the necessary transformations successfully, there were still improvements that could be made to create a true generic transformation language and engine to be used in OPC UA server aggregation.

The purpose of this thesis is to research model transformations through how similar models have been transformed using other technologies and techniques. This information is used then to design a language for transforming OPC UA address spaces to any arbitrary form the user desires, to generate an aggregate server which serves as an interface for multiple underlying servers.

1.2 Research Objectives

The primary goal of this thesis is to design, implement and evaluate a rule language and an algorithm for transforming an OPC UA address space into a different OPC UA address space with a subset of the value nodes of the original address space. Because

such a system has not been publicly done before, discovering the requirements for the language are also part of the research goals. The design is evaluated through three predetermined address space cases in chapter 9, which are the following:

- Simplifying an artificially created address space
- Complicate the address space transformed in case 1 back to its original form
- Transform a real-life device address space

The first case case is a generated simple address space which utilizes several features in OPC UA. The most relevant parts are the use of multiple different types of nodes, custom node types, references, attributes and properties. The second case is to prove that the transformation can also be reversed using the same language, and that new information can be created and added into the address space as well as reducing the complexity through removing levels of information. The third case is a real-life address space example, which follows external requirements for the transformed address space.

1.3 Research Methods

This thesis is follows the design science framework. Design science is defined as a framework, where the designed artifacts are used to solve a well-defined problem, or to improve the functionality of the designed artifacts themselves [21]. Design science is well suited to solving incrementally built systems, algorithms and developmental problems, such as the problem described in this thesis. The artifact produced here is the designed and implemented rule language, transformation algorithm and parts of the aggregating server necessary to use the new language to perform the transformations.

There are four steps in the design science process: requirements definition, design, implementation, and evaluation [21]. Because this type of a transformation process has not been previous done to OPC UA address spaces, the very first part of the process before the requirements can be fully defined, is to do extensive literature research on other rule languages used in similar scenarios and environments. After the requirements have been defined, the previously mentioned artifacts are designed to fulfill them. They are then implemented and tested against the requirements to see how well the design stands up to the initial expectations. Based on the requirements, a set of test cases will be chosen to test out the requirements, and the implementation must successfully transform all of the test cases to pass the inspection.

This thesis is mainly focused on the requirements and design parts of the design science process. The implementation and experimentation of a rule language does not bring much new information aside from the direct information of how well the design part is working in comparison to the requirements. Both of the identified requirements and presented design, however, are new information that can be used to further develop the OPC UA technology and its uses.

1.4 Thesis outline

Chapters 2 through 5 introduce the technology framework and literature research done on the subject of this thesis. They establish the base on which the entire design is done on, and provide information on both the OPC UA technology and model transformation in general. This information is then utilized in chapter 6, where the requirements for a rule language and transformation engine for OPC UA address spaces are defined.

Chapter 7 is the most important chapter in the thesis, because it describes the design of the language and engine used in the transformation process. In chapter 8 the design is implemented, and tested out against three test cases in chapter 9 which have been designed beforehand for this specific purpose.

2 OPC Unified Architecture

In its core, OPC is a communication framework designed to be used in a variety of systems, where communication between devices or machines is a priority. The most common applications include SCADA and HMI systems, but it is also known to be used for abstract information, such as customers or product data or recipes. It is capable of transferring information from devices and sensors, such as engines, manipulators and optical readers, but it can also be used to control those devices through writing new control variables through the OPC system.

OPC UA is built on a client/server architecture. The servers of OPC UA expose data from the lower levels by presenting the data in a model structured from layers of complexity, starting from the broad point of view, such as a complete factory, and ending in leaves of values, such as those of single sensors and manipulators. This model is called an address space, which will give each point of data and each layer a unique identifier so they can be accessed directly without the need of always starting from some entry point. The clients of OPC UA are used to access the servers to read and write the data on each node on the server, subscribe to data changes, or make alarms and events [26]. These are only some of the possible uses of OPC UA, and even more are discovered as the specification expands.

2.1 OPC Classic

OPC Classic is the first and most popular iteration of the OPC family. It has been widely applied in many fields of engineering, where data collection is necessary. OPC has been expanded with multiple additions, most of which have been turned into an integral part of OPC UA. These expansions are Data Access, Historical Data Access, Alarms and Events, XML-Data Access, Data eXchange, Complex Data, Security and Batch.

The biggest issue with OPC Classic is that it is dependent on Microsoft Windows platform as it uses COM and DCOM technologies in its communication. The old implementation is also limited in multi-threaded processing capabilities and security of the system, which are a growing concern among technology developers and companies today.

2.2 OPC UA

Compared to the classic OPC, OPC UA has improved on the issues associated with the Microsoft platform dependency by completely rebuilding the technology from the ground up, selecting multi-platform and independent commincation frameworks and architectures instead.

OPC UA provides the developer a modeling-based platform in address spaces to start off the development of the servers. Models are a popular method of visualizing objects and consepts [28], and allow the users to visualize the data for easier human consumption while still being flexible and complex enough to let nearly any type of data to be displayed through the OPC UA servers. Among the features [24] which

make OPC UA better than the OPC classic systems are also that the OPC UA models are extensible, and they allow meta-modeling of the system and data, and the entire system is server-oriented, which makes the entire architecture modular and flexible [8].

Other improvements of UA include being server-oriented, extensible, and enabling meta-modeling of the system, which were restricting factors in the development of OPC classic system.

With the development on OPC UA, it has started to replace the classic OPC among most developers, and is attempting to replace old OPC solutions with the new technology. The active development of OPC UA has enabled it to expand to a variety of programming languages, some of the most popular choices being C/C++, Java and .NET. The development of the language-specific development kits has been delegated to specific companies by the leave from OPC Foundation. OPC Foundation [2], Prosys [3], MatrikonOPC [1], Softing [5] and Unified Automation [6] are all developing and maintaining separate SDKs and other OPC software products, all working on one or more of the aforementioned programming languages.

The most important feature relevant to this thesis added by OPC UA is the address space. It is the feature which enables the transformations to be created using generic patterns [16], as the matching process will be able to work with much more information than just a single string representing the name or identifier of a node. Patterning will allow the transformationg engine to work efficiently and the interface to be simple to use for the operators.

It will also allow the transformation to work on a meta-level instead of having to be tailored specifically for each solution, which means that common rules can be created for similar systems and distributed by the developers.

2.3 Address space

An address space is an information model of all the data stored on an OPC UA server [25]. The two most important parts of an address space are nodes and references. The most common way to visually represent an address space is to build a tree structure from nodes and their hierarchical references. As the entire address space is accessed through a single entry point, the root node, all nodes and references are supposed to connect to it through other nodes and references. A single node or reference can exist without any access points, but can prove to be quite useless, as it cannot be browsed to from the entry point, unless you already know the NodeId beforehand.

Nodes are the core elements of an address space. A single node can contain multiple references, attributes and values, and is meant to represent a single entity in a system. It can range from a complex unit, such as a whole machine, to a single data point, like a sensor. When a node represent a larger machine, it itself will not hold all of the data values the system can output, but has references to each sensor and actuator as a separate node in the address space.

A reference is an element which links two nodes to each other. A reference should always contain a source and a target node, and the type of the reference designates a

direction for it. A reference can also be bi-directional. Using the references correctly, a very complex mesh of connections can be built to represent each level of technology in a complete system. This enables the user to view the address space as a kind of a tree structure of each layer in the system.

Attributes are used to describe a node more specifically. All attributes are always significant by themselves and denote a single quality of the node, but when combined, they represent every piece of information available on a node, describing even the smallest pieces of information. Nodes may contain any number of any types of attributes, but they always share a few common attributes designated at the OPC UA specification [24]. These attributes are displayed in table 1.

Table 1: Common attributes of OPC UA nodes

NodeId
NodeClass
BrowseName
DisplayName
Description
WriteMask
UserWriteMask

The most important attribute among all those available is the NodeId, which is used to identify a single node on the server. It has to be unique, and combined with the address space identifier, it creates a universally unique identifier for a specific node in an entire OPC UA system containing multiple servers and clients. Knowing the NodeId of a node you are trying to access lets you skip entering the model through some entry node, and directly accessing the node and it's attributes and values.

NodeClass describes the type of the node, be it Object, Variable, Method or View. Objects are containers for Variables and Methods, both of which can only be contained in an Object. Objects themselves can't hold values directly, but it can be be constructed to hold any type of values and metadata related to that node through connected Variables. Methods are similar to the methods used in programming languges: they are functions that can be called to execute a procedure with some arguments. Views are meant to reduce the amount of data visible to the user, by creating a limited set of nodes to be shown. This is useful for example to collect only relevant nodes from among thousands of data nodes available on a server.

BrowseName, DisplayName and Description are attributes which are used during browsing to distinguish nodes from each other when being displayed by a human user, or to recognize a specific node from a group of nodes of the same type and attribute information. The values of these attributes can be mostly arbitrary. In the OPC Classic, the names of nodes held a much more important role, containing all of the recognition information, but the modeling in OPC UA has moved the focus more towards the type and reference information.

WriteMask and UserWriteMask are used in preventing or allowing the access to all of the information on a node, either on a generic level or user-specific level

respectively. They are not relevant to the transformation process, except that the transformation system should always have full access to every node on the server.

3 Information model

An information model is a model used to represent the structure of information in a specific domain or system [19]. It is often structured as a tree so that all items in the model have a parent or some other type of a superior, but they can also contain cyclic references, which means that the model might not have a clear parent-child relationships unless specifically stated by the references themselves. The main pieces of an information model are types, objects, relations and properties, but some information models also contain behaviour information or other more specific information relevant to that model only.

The type information is a meta-level information available to show what constraints each different object, class or property must abide by. A type can restrict or enable the item to function in a very precise manner. The definition of the type contains all of the relations, properties and values that must be connected to the typed node to construct a valid model.

An object is the central piece of information on a model, which connects to all the relations and properties relevant to it. With the help of multiple relations to other objects, they can be used to construct complex structures, such as complete machines. Using them in a tree-like model, the machine itself can be represented as one object, and all the pieces in it are other objects with a parent-like relationship to the machine object. An object itself cannot contain any values, and thus even the smalles pieces of information must be contained within an object, which has a property with the desired value.

Relations are the connections between each object. A relation can be of any type of information binding the two objects together, such as structural (owned by) or informational (is affected by). The type of the relation describes how the two objects interact with each other in the domain or system. Relations may require a direction to be defined in the relation, for example the previously mentioned owned by relationship must define the owner and the owned object, but other times a relation is between similar objects and does not need the direction, such as a power line connection between two machines.

Properties are the holders of the actual values inside the system. Objects can contain multiple different properties, but the values must always be contained within a property, and no other entity in the model. Properties will also need a type or a name to properly define the data contained within to make sense in the model.

Behaviours are the actions, functions and methods an object can perform. These can refer to programmatic features, such as direct method calls, or they can be meta information, which simply describes the actions the object is capable of doing.

3.1 Address space as an information model

An OPC UA address space has all of the features an information model requires. It uses nodes as objects, references as relations, attributes and properties as properties and it even has types and behaviours directly. As a bonus, the specification of an information model does not restrict the model to be built in a tree-type structure, and

OPC UA address space uses this to its advantage: it can contain cyclical references, which means that any node can have a reference to any other node in the same address space. This does cause its own problems, which are discussed later in chapter 5.1. The address space therefore contains all of the elements specified to be in an information model, and therefore any language designed to work on a generic model that reflects the features of an information model can be modified to work on an address space model without too much complexity.

4 Transforming models

Model transformation is the process of taking a model, called a source model, and changing it in some way to create a completely new model, called a target model. The transformation process will use all of the information available in the source model, and based on how the user has instructed the transformation to be handled, will construct the target model, while making the desired adjustments. The changes made during a model transformation can be relation changes between objects to restructure or reorder the entire model, or they could combine the values of two properties to create aggreagted values in new objects. They might also create new objects or remove them to complicate or simplify the model as desired. As an example, the figure 1 shows how nodes from both Category1 and Category2 are sorted under New category in the target model, and the categories are completely removed from the node Value2. The transformation would shift around all the data nodes, but keep the information intact in each node, and the node's attributes.

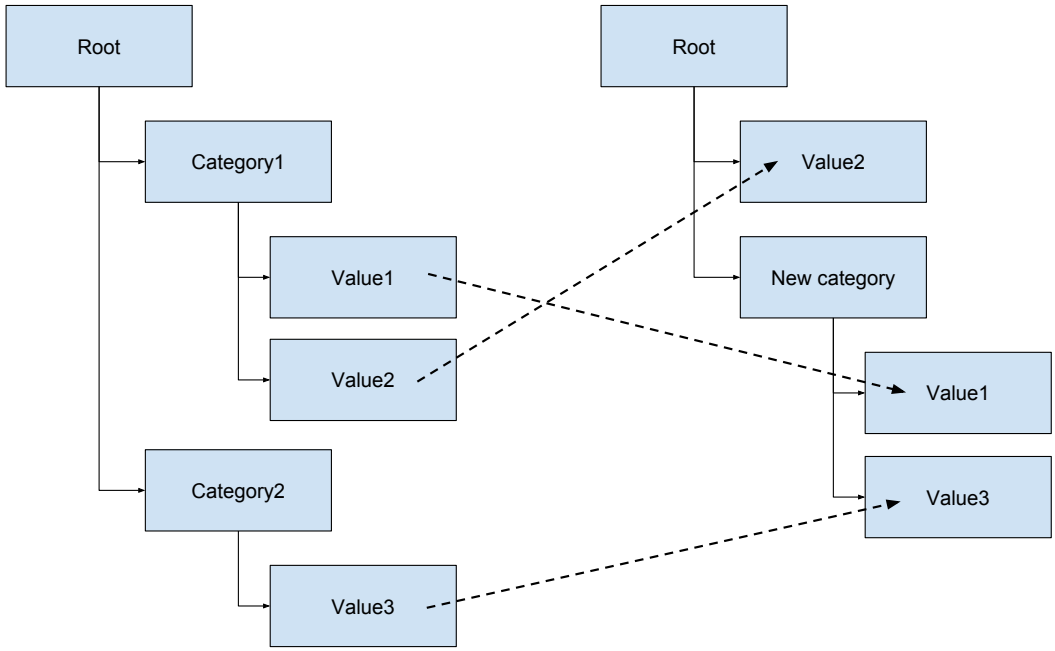

Figure 1: An example of a model transformation process. The left model is the source model, and the right model is the target model. The dashed lines are the rules that will generate the newmodel and create connections between the nodes to transfer data changes from the source to the target.

Model Driven Engineering (MDE) is a popular environment that utlizes model transformation as a tool [28]. In MDE, model transformation is used as a part of the software development process, where all of the produced domain models must be consistent with each other. The transformations are there to ensure the consistency, reducing the amount of problems arising during the process of turning the models

into actual software. A standard language for MDE transformations is the QVT (Query/View/Transformation).

A language which is very powerful in representing tree models is eXtensible Markup Language (XML). XML is a general-purpose language, usable in multiple different scenarios. It has it's own transformation language, Extensible Stylesheet Language Transformations (XSLT) [10], which is used for transforming any XML derivative sheets into any other XML derivative sheet, or even other formats, such as PDF.

Resource Description Framework (RDF) and by extension Web Ontology Language (OWL) [23] are used to model web ontologies. An ontology is a set of descriptions, that construct the properties and relationships between resources in a domain, and is used to describe the categories and types, but not single instances of those categories. An information model is very similar to an ontology in the sense that they both are structured using information about relations between items in the model, but an information model handles the single instances directly unlike ontologies.

A few examples where ontologies are used in transformations are in combination with MDE [13], where an ontology is used as an information model parallel to the common models of MDE to create a comprehensive model of the entire system and it's data. What this means is that the ontology helps to define better the instances of data described in MDE, and assists the transformation process by linking otherwise unattached pieces of information by setting them in a pattern specified by the ontology to be more easily recognized by the transformation processes of MDE. Another case would be where an ontology is used as the basis for creating SQL database tables [7], and the transformation is used as an automated process for the creation and storage of any type and size ontologies in these tables. The ontology, similarily to the previous example, helps to structure the information and create recognizable patterns so that the data in the tables will follow a cohesive real-world structure defined in the ontologies.

4.1 Model Transformation Language

A crucial part of model transformation is the language used to describe each unique transformation. All transformation languages can be considered to be either internal or external to the transformed system [22].

An internal DSL (Domain Specific language) is a language, which uses a host language, most likely the one used to represent the system being transformed or the target system. It is tied closely to the host language in both expressiveness, complexity and features. It is much less prone to mistakes and mismatches representing the system, and the implementation work can make use of the existing tools in the host language. The downside of using an internal DSL is that it requires domain knowledge of the host system from the user.

An external DSL refers to a language, which is independent from the original system. Building an external DSL requires much more additional work than an internal one, requiring the developer to also produce tools to write, manipulate and run the transformations through parsers, compilers and interpreters.

5 OPC UA transformation

Because there has not been much research done for OPC UA address space transformations, the mechanics for performing these transformations must be derived from other transformation and mapping methods and technologies.

There are some tools currently in the market that are able to perform some type of transformations for OPC UA address spaces, but most of those are simple, and mostly only transfer the original address space as a whole to the target server. One example of such a tool is the OPC UA Historian [4]. It, however, is capable of using an SQL database as the Historical Data Access (HDA) service of the OPC UA server, and this could be considered a transformation process for historical data points.

Grossman [18] suggests in his paper an automatic process to map aggregated multiple servers to one. It is not transformation in the sense this paper is looking for, but using OPC UA itself as the platform for holding the rules gives some interesting basis on how the rules could be stored as a sort of an internal DSL using OPC UA nodes and types to identify the LHS of the rules. This system, however, does not provide much flexibility in the construction of both sides of the rules, and as such is not a very well positioned candidate for transforming arbitrary OPC UA address spaces to any other arbitrary address spaces.

In Elovaara's work [14], he continues on Grossman's proposed aggregating server. He has created a basis for a rule language for transformations on Java using a generic rule language tool, Drools. He has recognized that an external language is required to make the transformations work, and has made a functioning system that performs some transformation operations, but only in his limited requirements environment. As such, it will not work as a complete transformation language, because it is not capable of doing arbitrary transformations, but provides much insight into actual working transformation language development progress.

5.1 Cyclic references

As the OPC UA address space is not structurally defined except by its references, a way to browse through the entire address space must be well defined. The most common way to display an address space visually is through the "HasComponent" references between UA nodes. It, however, may not reach every node on the server, as it is not required for a node to be exactly behind such a reference. Any reference can be used to connect a node to the address space web. Cyclical references and not all nodes being behind similar references create issues major which must be solved by the mapping tool.

A simple way to retrieve all nodes from the server is to simply browse all references, and storing each Node or NodeId in a storage, and avoiding browsing those nodes again. This may use up a lot of time though, as some OPC UA servers can grow to be over a million nodes large, so alternative ways to reach the desired nodes should be considered.

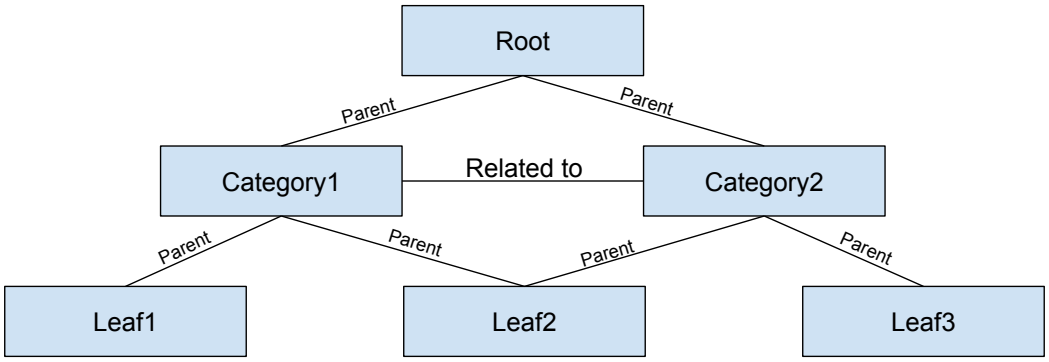

Figure 2: Cyclic references in an address space. Leaf2 also displays how a node can have multiple parents in an address space.

5.2 Reference preservation

When restructuring an address space, considerable care should be taken to preserve every possible reference between nodes being transformed from the source to the target server. This becomes increasingly difficult when performing either of the following actions during the transformation: attempting to reduce the amount of information by removing levels of complexity from the structural model, and how the references of the removed nodes should be displayed in the target model; and when using aggregate nodes, such as sums or averages, and how references from the original nodes are handled.

5.3 Type inheritance

OPC UA enables types to be inherited through other types. The LHS rules should be able to recognize nodes with subtypes when the nodes are being processed by the transformation engine. The engine should also be able to create supertypes for the target server without direct references to those types when a node with a subtype is transformed.

6 Requirements

The requirements for transforming OPC UA address spaces have not been previously defined, so this chapter focuses on their discovery. The requirements have been found through observing other rule languages, such as XSLT [10], XPath [11], ontology mapping [9] [13] and ontology transformations [31]. The study on regular expressions in query languages has also brought insight into using text-based rules to parse model data [15].

6.1 Rule language

A rule language [12] is a set of parameters that define a language constructed and formatted to be machine-readable, and is capable of representing both the source and target domains of a single transformation process. The definition of a rule language must be described in such a manner that it can be implemented in either its own environment, such as the host language or technology it has been designed for, or in a generic environment, in which case it must be adaptable to any language or technology which wants to use it for its own advantage [30]. There are two things a user needs when creating rules using a language specifically tailored for that purpose: source language knowledge, in this case OPC UA address spaces, to properly express the models using the rule language, and business knowledge, to create rules that perform transformations with a clear purpose [20].

A rule consists of two basic parts, structured in a way represented in figure 3: the left hand side (LHS), also known as the conditional part, and the right hand side (RHS), also known as the consequence part. LHS refers to the part of the rule which is the source of the data to transform, identified by all of the parameters in the LHS rule. The RHS, on the other hand, is the target data, where the information will be transferred to.

<Condition> = <Consequence>

Figure 3: The basic structure of a generic rule language

In this thesis, the rule language has been designed to be implemented into any environment that is using OPC UA, and is only required to be able to handle OPC UA address spaces as the input and output of the process. OPC UA address space model is a very generic platform to build on, and can be very complex because of this. Many portions of the model can be customized to the user's needs, and the rule language can not rely on the OPC UA specifications to know all the information a node can contain as a result. However, as discussed earlier in the OPC UA chapter 2, a node has a set of attributes and references common with all OPC UA nodes. Among these are the type information through a HasTypeDefinition reference, BrowseName, DisplayName, NodeId, NodeClass, Description, WriteMask, and UserWriteMask attributes. Out of these, NodeId is useless for attempting to recognize nodes through patterns, because it will be unique for each and every node in the address space. NodeClass, when combined with the type information from the HasTypeDefinition

reference, is undoubtedly the best way to recognize a node based on the model, especially when adding the references leading to and from the node. Unfortunately, because of how classic OPC handled nodes and the lack of extensive modeling, BrowseName and DisplayName will likely still be used in companies to refer to nodes in a system, most likely through naming and numbering the nodes in a supposedly recognizable pattern. Thus, the names, class and type information will be the key information used in node recognition.

When comparing type information, it is imperative that the rule language, or more specifically the transformation algorithm, is capable of handling the inheritance of node types present in OPC UA. When a rule is written to search for a specific type, the interpreter must be able to find not only the exact reference to the given type, but also to all types that are inherited from it. A rule could be written to find all nodes that are of the BaseObjectType, which is the root type for all object types, and therefore should match with all objects in the address space that are typed.

In addition to those, some cases require certain specific attribute values to identify the required nodes from a group of similar nodes. Because of this, the language must be able to compare specific attribute values, both the common attributes and any custom properties generated by a user for each node, to constant values defined in the rule. An advanced method instead of simply comparing an attribute to a constant value is to also compare two attributes between themselves, or comparing the attributes of two different nodes. This also could also include the common logic operators such as “NOT”, “AND”, “OR”, “XOR”, “NOR”, “NAND” and “XNOR” between each attribute comparison for a node. This could even be expanded to what type of references a node can and can not have, but all of the advanced parameters for rules have been left out of this version of the rule language design.

OPC UA is also a model with cyclical references, which means that simply by following all the references leading away or to a node, the same node can be found again without retracing the same references already used before. This makes OPC UA address space a web model, which unlike a tree model, must be browsed with particular care not to end up in an infinite loop during the browsing process. The rules should be able to be written in such a manner, that the algorithm processing the rules and address spaces does not get locked in an infinite loop simply because of the structure of the rules.

The language must be designed in a way that a single rule can be read and understood by a user without the need for a specific software to display it [29]. It is imperative for the spreading and usability of the language to be viewed in such a way, and computer-only readable languages have died off because of this very reason, even if they have otherwise completed the requirements for the task they were designed for.

Care should be taken to make the language Turing complete [22]. Turing complete as a description of a rule language means that it is capable of performing computational operations similar to a programming language. This means Internal DSLs have the benefit of being Turing complete if the host language already is, but external ones need to be developed in a way to incorporate this feature.

6.2 Transformation engine

Transformation engine is integral part of the aggregate server software. It is the processor for the rule language described in the previous chapter. The transformation engine is called when a new source server is added to the aggregate server to be processed, but is no longer needed by the server program after the transformation is complete. It is strictly there to handle the transformation and establish the links between the nodes in different servers.

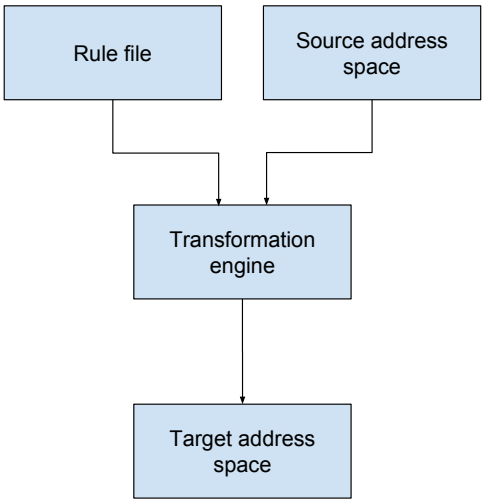

Figure 4: How the transformation engine process interacts with the address spaces and rules

The first role of the transformation engine is to interpret the rule language. This means that the engine can read a single rule and turn it into a form easily handled by the software in the second part. For the greatest efficiency, the new form should be able to match with OPC UA objects directly.

The second role of the engine is to match each rule with appropriate nodes in the source address space (figure 4). While in OPC UA nodes can be connected with any type of references, this work will assume that every node relevant to the transformation process can be reached using HasComponent or Organizes references. Those two references have been selected based on the cases described later in this work and on a few real-world scenarios (Appendix A) used as source material. These references also have the inherent ability of not causing loops in a physical representation of an object being modelled.

Matching the entire rule means finding a chain of nodes connected with HasComponent or Organizes -references, that all have the types, names and attribute values specified in the rule for each node in the exact order also specified in the rule. While OPC UA address space is a cyclic model, meaning that the references can exist between any two nodes, the references do have directions. When matching nodes, if a node is before another node in the rule, it is considered to be the source node of a HasComponent or Organizes -reference, which will reduce the number of

possible connections drastically. When developing further, the language and the engine should move on to handle any type of references, and in any direction, to even further pinpoint the required nodes for each rule.

The engine should not restrict the transformations in any way. Specifically, this means that the engine should allow each rule written by the user to be performed, even if it means duplicate nodes would be created in the target address space. Each rule should be processed explicitly as written, and no part should be left undone, even if it means some nodes would be left useless because of this. This might be the intended behaviour by the writer of the rules, and thus should not be prevented by the program, even if it might seem pointless from the program's point of view. The only restriction the engine should place on the source address space is that it should browse each node only once. Otherwise the cyclical nature of OPC UA will leave the engine browsing the address space indefinitely, especially when using any type of references for browsing instead of just the previously specified HasComponent and Organizes -references.

When the engine is performing the transformations for multiple levels of nodes at one time, it should take note of every other type of reference originating from the nodes being transformed, and attempt to recreate those connections in the aggregate server address space if possible.

6.3 OPC UA

The OPC UA address spaces only have one requirement that needs to be fulfilled: it needs to follow the OPC UA specification strictly. For the purposes of this work, it means that each node has to have a HasTypeDefinition reference, which points to a type node on the source server. Without the types, recognition of the nodes falls to only the name and attributes, which are much more likely to be similar between two very different nodes.

For the purposes of actually performing transformations on each and every node on the source server, it is necessary that each node that the user wants to transform can be reached only using HasComponent and Organizes -references. As explained before, these references have been selected based on the cases described later in this work, and a few real-world scenarios (Appendix A).

6.4 Aggregating server

An aggregating server for the purposes of this work is a server that is capable of linking OPC UA servers together by passing through all OPC UA data requests and can update node values in real-time. The most important data manipulation requests for such a server are read, write, and subscribe actions described by the OPC UA specification [26].

The aggregating server will need to be able to browse an external address space, and perform the transformation process after it has been properly implemented. It needs to provide all of the necessary address space and rule information from other parts of the software and its components to the transformation engine to properly

integrate each part to work as a whole. It must be capable of holding the new address space, and create links between the nodes on itself and the source server based on the information provided by the transformation engine.

The server must be capable of performing all of the above in a reasonable time, so as to not cause delays or missed information after the transformation and node linking has been performed. While such delays often are of no issue, the biggest bottleneck should not be any of the internal workings of the aggregating server, but rather be on the connection delay between the servers themselves.

7 Design

The design of this work focuses heavily on the rule language. The development of the engine itself is rather straightforward, as it does not perform any truly complex work alone, but rather must work as instructed by the rule language. Because of this, the most important part of the work is the functionality of the rule language, and how it is interpreted by the engine.

7.1 Rule language syntax

The rule language syntax was mostly influenced by XSLT [10], BNF [17] and regular expressions available in most computer languages. The rule language is intentionally not based on any programming language. While it would have made sense for a programmer, the language should be free of a specific programming language to allow the greatest amount of flexibility between different implementations of the transformation engine for different platforms.

There are patterns for the language syntax which can be used during the language development to help it achieve easy understandability and help the user to adapt to use the new language. A well known pattern for this is the Backus–Naur Form (BNF). It has generic structures for both the LHS and RHS of the rule [12], and for each element on each side, and has the benefit of being completely context-free of the environment. BNF has later been extended to Extended Backus–Naur Form (EBNF) and Augmented Backus–Naur Form (ABNF), both of which are more expanded in features, but are meant for more specific platforms and languages. BNF has other alternatives for different levels of abstraction, one of them being Van Wijngaarden grammar.

The cases in chapter 9 and a real-life scenario of an address space in appendix A and B provided the information as to which parts of the node were most relevant for recognizing a single node from the entire address space. Unsurprisingly, the meta-data of the type and connecting references were the most powerful tools for this, but the name or a part of the name was also useful in some cases, as the nodes still followed a pattern of static name plus numbers to discern the different instances of the same model from each other which has been the way OPC classic was used. For the purposes of the cases in this work, the “hasComponent” reference was selected as the central reference type, which could be used to reach most, if not all, nodes in a single address space. The only other competing reference type was “organizes”, which had a lot more use in the real-life scenarios, could be easily changed to be the central reference type.

To distinguish the difference between type, name, attributes and transformation reference information for the recognition pattern, specific symbols were selected to be used for each different piece of information. Having to select these for the rule pattern does currently cut off these symbols from being used in the naming of the types, nodes and attributes, but similarly to regular expressions, an escape character could be used. There are two major driving forces behind selecting these characters specifically. First, the characters used are very unlikely to be used in

the types, names, attributes or values of the nodes. Secondly, the method in which the characters are used is familiar from other programming related notations and languages, such as the regular expressions mentioned earlier.

For type information, the '[' character starts the type, and the ']' character ends it. For the name information, no characters specifically start or end the name, but instead any letters not surrounded by any of the other special characters are considered to be part of the name. For the reference information, the character starts it, and goes on until the next special character or the end of the rule. In practice, the reference ends on the beginning character of the attribute information '(', the reference attribute specifying character '@', the character denoting the beginning of a new node '/', or the end of the rule. For the attribute information, all of the attributes are contained between the '(' and ')' characters, and each individual attribute is separated by a ',' character. In each attribute, the '=' separates the attribute name and the attribute value. The attribute value can also contain a reference to a specific other node from the LHS of the rule. In such a case, after the separator character '=', start the sequence by a reference character '#', input the reference name, and follow it up with a reference attribute character '@', after which enter the attribute name. To combine multiple attribute references and arbitrary text to a single value, even from different sources, you can use the in-attribute reference wrappers, '{' to start it and '}' to finish it. The text between the wrapper characters can only contain a reference character '#', reference name, reference attribute character '@' and the attribute name. It is intended to be used to resolve each reference separately, and combine them with any arbitrary text between each wrapper. Any text between any two complete wrappers is considered to be static.

These characters can be used in the LHS of the rule to represent conditions when a node is matched, and also on the RHS to represent values to be placed in the newly created aggregating node.

Every parameter in the rule is optional, and can be left out completely. While it is completely possible to leave out every parameter in the rule, it is in most cases quite useless. Sometimes this is useful, however, if any node is valid as a match for a part of the rule. The order of the parameters has to follow the same pattern:

1. Type
2. Name
3. Reference information
4. Attributes

A final parameter in a rule is the type of the entire rule. Two types were selected for the purposes of this work: Copy and Deep copy. A copy simply copies the matching node based on the rule, while a deep copy creates a one-to-one clone of the entire tree of "hasComponent" references and nodes under the matching node in addition to copying the matching node itself based on the rule.

7.1.1 Rule file

The rule language is designed to be used in an external rule file. The processing engine expects the file to be injected into it by the user, and for it to work, it needs to contain all of the information about the rules, and the address space which the rules are designed to be used in. As the rule language is completely independent of a programming language, any person familiar with the language and the address space model can adjust the rules as they see fit.

As a rule file is not a part of the source code of a program, it can be adjusted outside of the program, and given as a parameter at the latest when the actual mapping is going to take place.

7.2 Transformation algorithm

The transformation algorithm starts off by browsing the nodes in the source address space to find the appropriate nodes matching the rules to begin the transformation process. As per the requirements of the OPC UA address space in chapter 6.3, the address space is assumed to be built out of folder nodes, object nodes, HasComponent references and Organizes references. While browsing, every other type of reference could be safely ignored based on these assumptions. In addition, property nodes are also ignored, as the properties are always attached to an object node, and therefore transformed alongside said node unless otherwise specified in a rule.

While the browsing could be started from the root node of the address space, browsing through types and server information would be pointless. The types are transformed alongside the nodes which reference them. Therefore the Objects-folder, situated right under the root node, was chosen as the starting point for the transformation algorithm. The algorithm starts off by browsing the nodes from top to down, stopping at each node to perform the following steps.

1. Take each rule that is appropriate for this address space
2. Evaluate each rule separately to this node using the following steps. If at any point the answer is negative, the rule does not apply to this node, and can be skipped.
    (a) Starting with the last node of the LHS of the rule, compare the current node to the values of the node in the rule.
    (b) Take each node that has a “hasComponent” reference to this node, which will be called “parent nodes”. Follow the rule one node backwards, and perform the same comparison between the parent node and the rule node. For each parent node that applies, save the reference information if required by the rule for further use.
    (c) Repeat step b. until we reach the final node of the LHS of the rule. For each complete path of parent nodes that applied to the rule, perform the transformation process.

The actual transformation is then performed for each complete path of nodes that applied to the rule. Each complete path creates a separate process of transformation with its own reference information for each node.

3. The transformation process starts by transforming the first node in the LHS of the rule:

   (a) Using the reference information, create a copy of the source node to the aggregating server, under the Objects-folder. If no reference information was given, create a completely new node, and if necessary, also a completely new node type if it was provided and not found on the aggregating server already. If no type was provided at all, instead create a folder type node. Then set the attributes specified separately in the node to their given values. Use the reference information to find the values from other nodes if necessary.

   (b) Following the LHS of the rule downwards, perform the step a. for each node in the rule, and create a "hasComponent" reference between each created node.

For further clarification, here are a few examples of the specifics inside some of the steps:

Step 2.a. A rule "[Type]Name" states, that the mapping engine needs to find a node that has the type Type, and the name Name. A node "[Type]OtherName" does not satisfy the rule, nor does a "[OtherType]Name".

Step 2.b. A rule "[Type]Name/[OtherType]OtherName" states, that the engine needs to find two nodes, with a "hasComponent" reference between them, with the node of type OtherType to be the target of the reference. A node "[Type]Name" that has an "organizes" reference to "[OtherType]OtherName" does not satisfy the rule.

Step 3.a. When a node following the rule "[Type]Name#Reference" is found and the complete rule is satisfied, the node is saved under the reference of "Reference". It can then later be accessed by the RHS of the rule creating a node based on the rule "#Reference", which attempts to find a previously saved reference with the name "Reference".

8 Implementation and experimentation

The implementation of the entire system is based on the original work of Elovaara [14] further building on the aggregating server implemented by him. The entire code is developed in Java, using Prosys's OPC UA Java SDK as the base for accessing the OPC UA address space and performing the OPC UA operations, such as read, write, and subscribe.

8.1 Rule manager

The rule manager is built to read and parse all the rules constructed by the user and turn them into machine processable form. The first step to the rule management process is to parse the text rule. The program attempts to find all of the special characters specified by the rule language, starting with splitting the rule left side from the right side. Due to time constraints the actual file parsing was left out from this version of the implementation, and the rules are inputted to the parsing program by manually inputting them into the code. This feature could be trivially replaced by a file reader though.

The parsing of the LHS and RHS differ very little, but have small nuances to them. Parsing the LHS produces a sequence of LHSRuleNode objects, which inherit the RuleNode object, to easily match with the UaNode objects discovered through browsing the address space. Each RuleNode contains the parameters given to a single node in the rule: name of the type, BrowseName, reference name, and any specific attribute values. LHSRuleNode object provides methods to help with the matching process used later in addition to the variables from the RuleNode object, whereas the RHSRuledNode has a method for attempting to find a node on the already transformed target server that would match the one being transformed perfectly. That method can be used to reduce the amount of duplicate nodes generated by rules transforming the same nodes most likely as the parent nodes of different leaves.

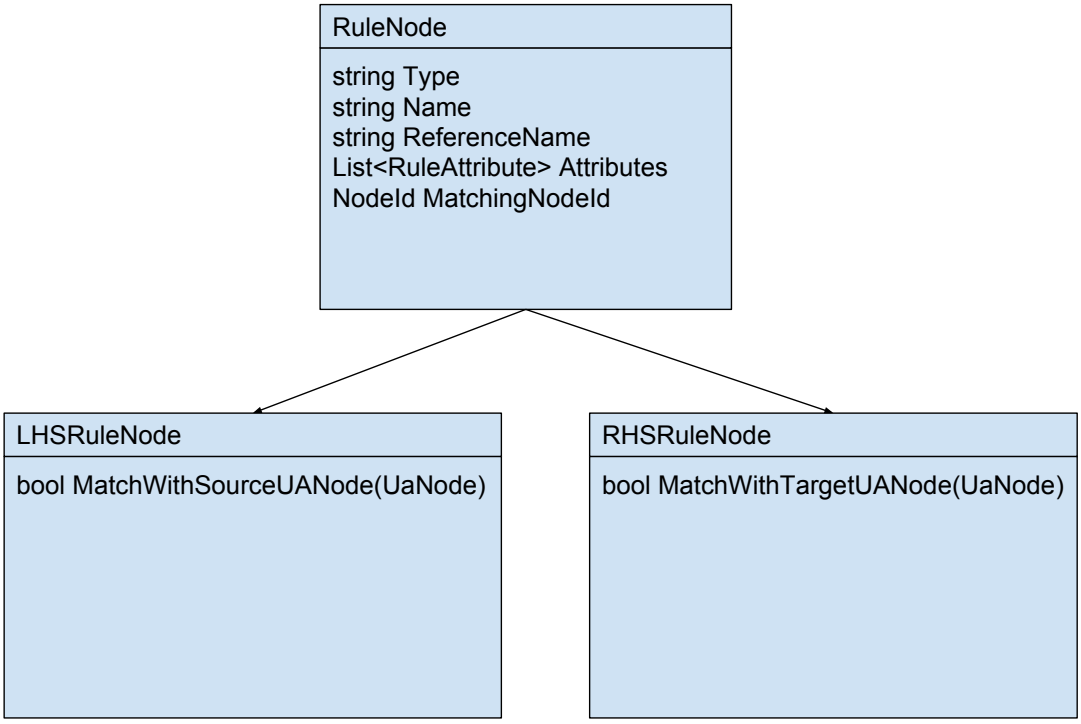

Figure 5: RuleNode class diagram

Likewise parsing the RHS of the rule produces RHSRuleNode objects, which inherit the same RuleNode object. RHSRuleNode provides its own methods for detecting matching UANodes in the target server, to prevent the creation of duplicate nodes that both have the same source node, and exactly similar RHS for creating a node in the target server.

8.2 Transformation engine

The original browsing algorithm in the aggregating server architecture implemented by Elovaara [14] was completely replaced by a new, simpler algorithm better suited for the new requirements. The algorithm attempts to browse all of the nodes in the source address space by starting off at the Objects folder that can be found under the root folder through an Organizes reference.

The biggest change from the previous one is that each node can match multiple rules. In the previous version, rules had priority, and each node ever only matched the highest priority rule it could. This did not allow users to create multiple copies of a single node in their aggregating server, and thus restricted the structure of the aggregating server. Allowing each node to match multiple rules lets future version of the rule language to create, for example, aggregating values from the nodes, or to organize nodes under multiple different folders, similar to views, but directly from the rule perspective.

The first step in the transformation process, presented in figure 6, as is discussed in the design portion of the transformation engine in chapter 7.2, is to check which

rules apply to the node. In the first part of this step, the transformation engine will fetch all the rules applicable to the source address space by comparing the address space name with the rule set names. All of the rules in the matching rule sets will be taken into consideration in the next part. The second part is to browse through the entire address space, and compare each node to each rule from the first part. During the browsing process, each browsed node is added to a list of browsed nodes. Before browsing a new node, the program check if the node is already in the list, and has already been browsed, therefore skipping that node.

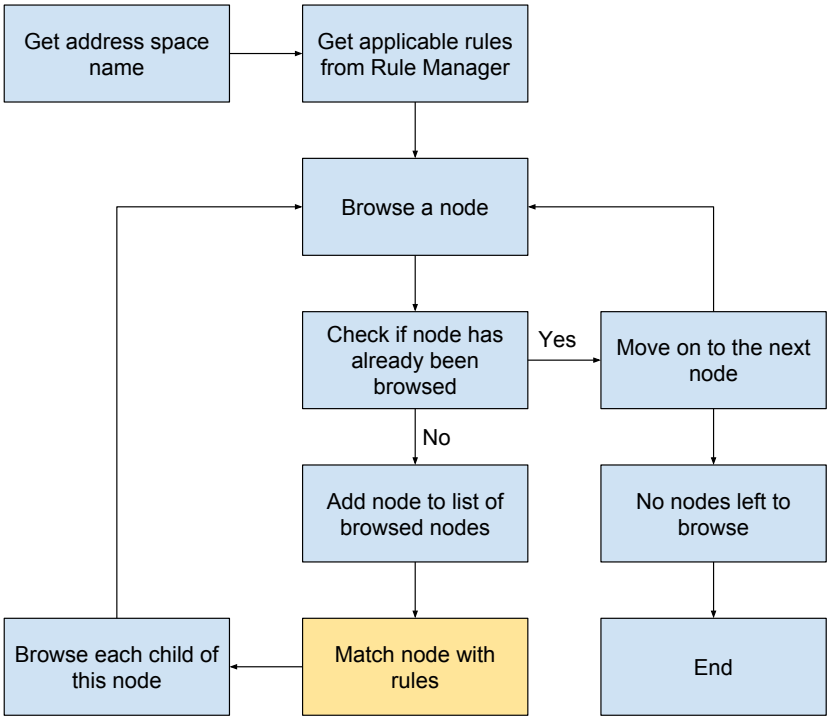

Figure 6: Browsing algorithm. The orange box depics the matching algorithm detailed in figure 7

To compare a node to a rule, as presented in figure 7, the engine starts off from the last node in the LHS of the rule, and compares it to the current node. Then, using the reference information, and specifically HasComponent references, it goes backwards towards the root node, comparing each node on the path to a corresponding node in the rule. If at any point no node matches a part of the rule, the current rule is discarded for the current node, and it moves on to the next rule. When there are no more rules to check, the engine moves on to the next node, and performs the same process all over to that node.

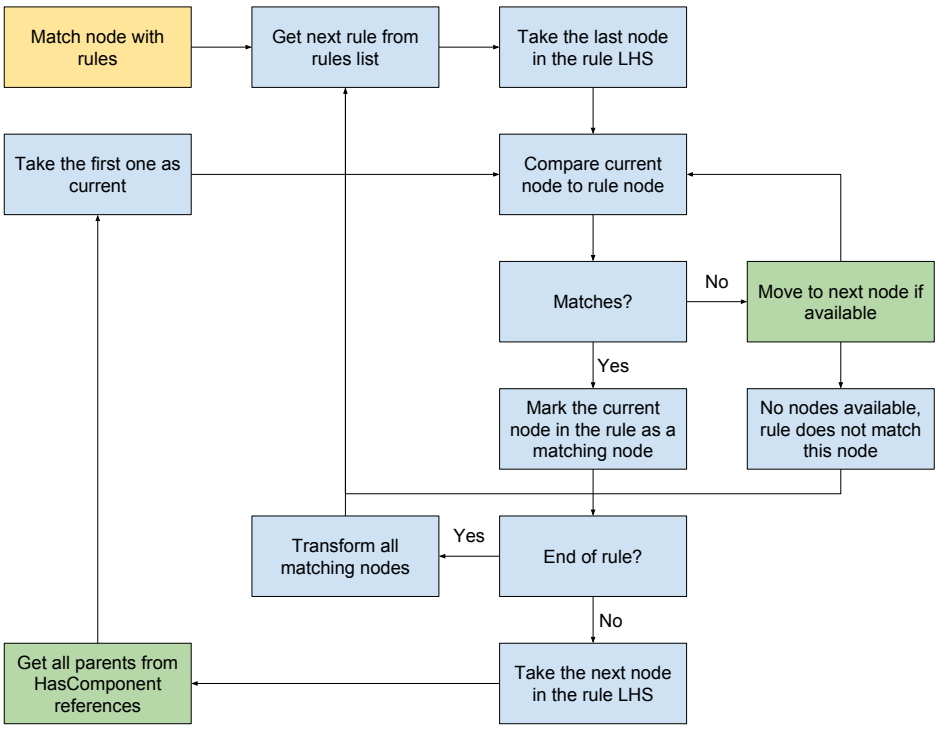

Figure 7: Matching algorithm. The orange box is the connecting node from the browsing algorithm in figure 6. The green box in the bottom left corner adds nodes to the node pool, whereas the top right green box

8.3 Aggregating server

The aggregating server architecture remains unchanged from Elovaara's [14] work. It implements the basic functions of OPC UA servers: reading and writing data, and subsciring to data changes. These features did not require any major changes when the old transformation engine was replaced with the new one, and thus remain in their original form.

The changes in the aggregating server only touch on the parts relevant to the transformation process. The complete works of transformation, rule file processing and storage in the system, and node creation on the aggregating server are remade to allow the new rule language to be used.

For each underlying server, the aggregating server creates a NodeManager, which contains all the information relevant to passing read, write and subscribe calls to the underlying server, and all the links between individual nodes in the aggregating address space and corresponding nodes in the underlying address space.

Whenever a read, a write or a subscribe operation is called on the aggregating server, a IoManagerListener will intercept this call. In the case of a read or a write operation, the listener will create a new call of the same type and send it to the underlying server that corresponds with the original namespace index and name based on the NodeManagers currently present in the aggregating server. The result of the

call is then passed forward as the result of the original request sent to the aggregating server. In the case of a subscription, the process is slightly more complex (figure 8). When a subscription call is intercepted by the IoManagerListener, a MonitoredItem is created, and the original call is subscribed to the node it was attempting to subscribe to originally. The IoManagerListener then creates a subscription to the underlying server again using the NodeManagers to discern the proper server to subscribe to, and then will connect the changes in this subscription to the MonitoredItem created earlier, which will write any data changes received through the subscription updates to the attached node in the aggregating server. The original subscription will then pick up these data changes just like a regular subscription should, and pass the data changes to the original subscriber.

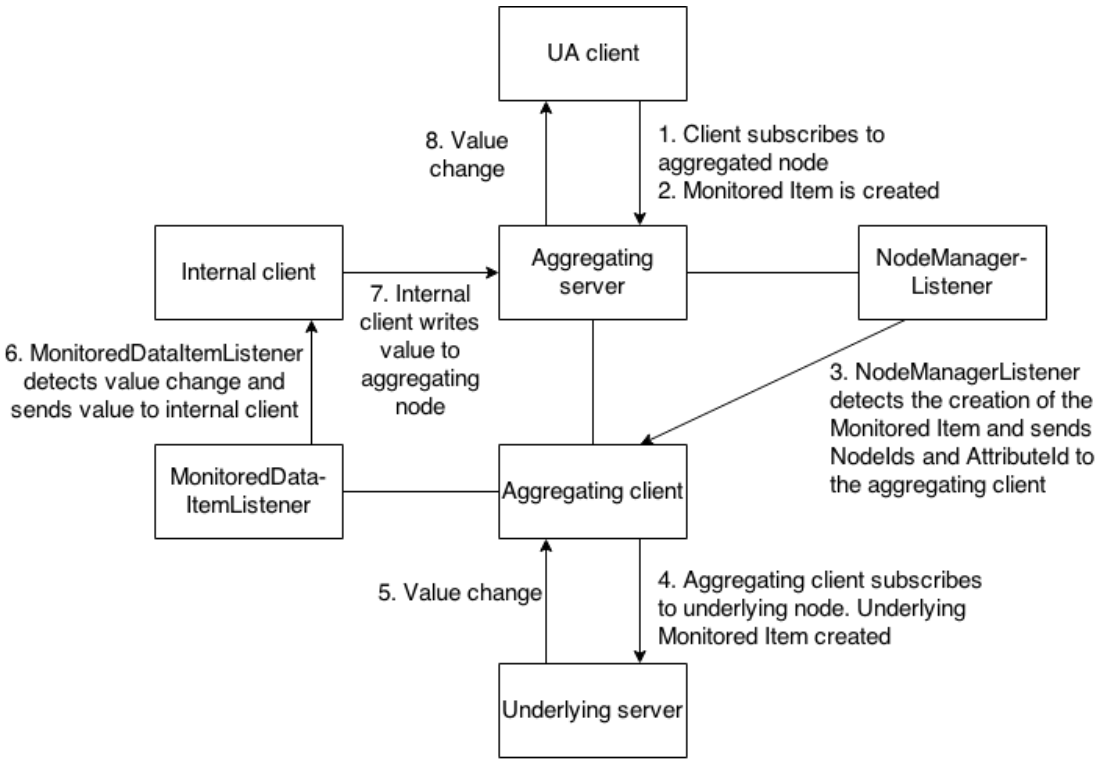

Figure 8: The relaying of subscriptions created by Elovaara [14]

9 Cases

The designed and implemented rule language needs to be tested against cases that either are or represent real-world scenarios. These cases attempt to capture the most commonly used environments and elements of a real OPC UA address space, and see how the rule language performs against those.

9.1 Boiler case

Boiler case is a simple address space of a boiler system with two pipes, a drum, and three controllers. It has been used in teaching material for OPC UA [24], and is therefore a reasonable starting point for a simple, and well known address space example for performing transformations on. Figure 9 displays the address space model structure and references between nodes. To complicate the scenario a bit, three instances of the same model have been instantiated on the source server: Boiler1, Boiler2 and Boiler3.

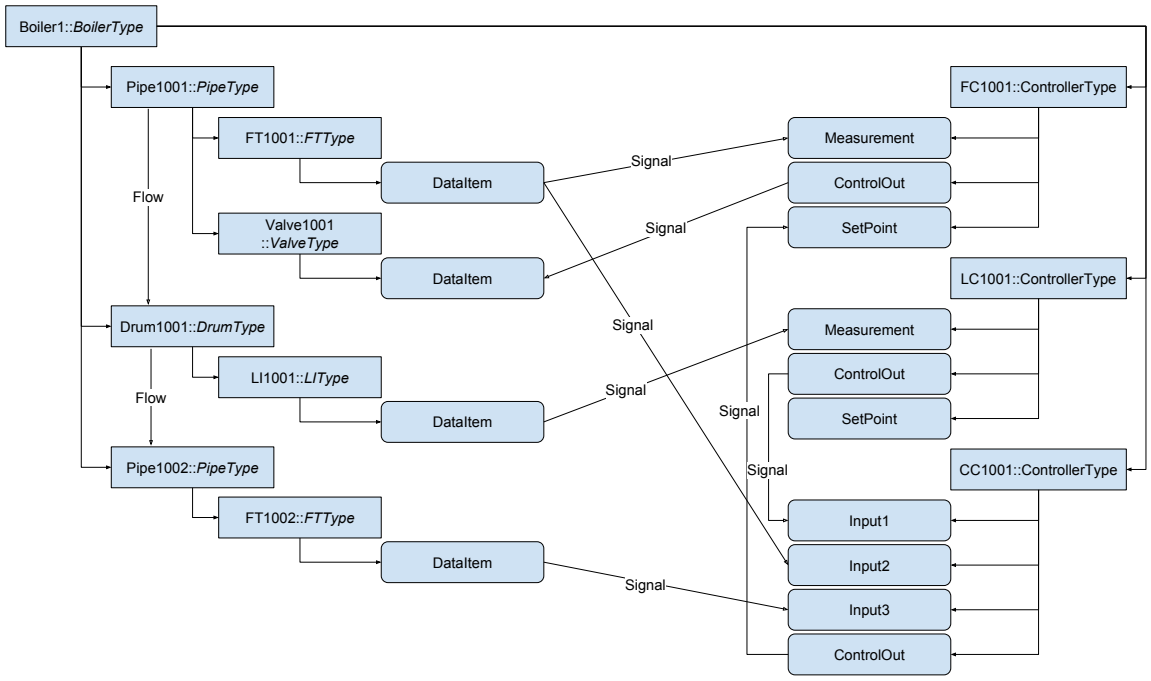

Figure 9: Boiler address space

The transformation process for this case started by first building the target address space to define what type of an address space is required as a result. From the difference between the models, the rules can be devised that will perform the required transformations.

The target transformed address space in figure 10 is a simplified version of the source address space. Layers between the Boiler node and variables nodes in the

pipes and drum have been removed, and the data nodes have been renamed to their immediate parent's names, to make the resulting nodes distinguishable from each other.

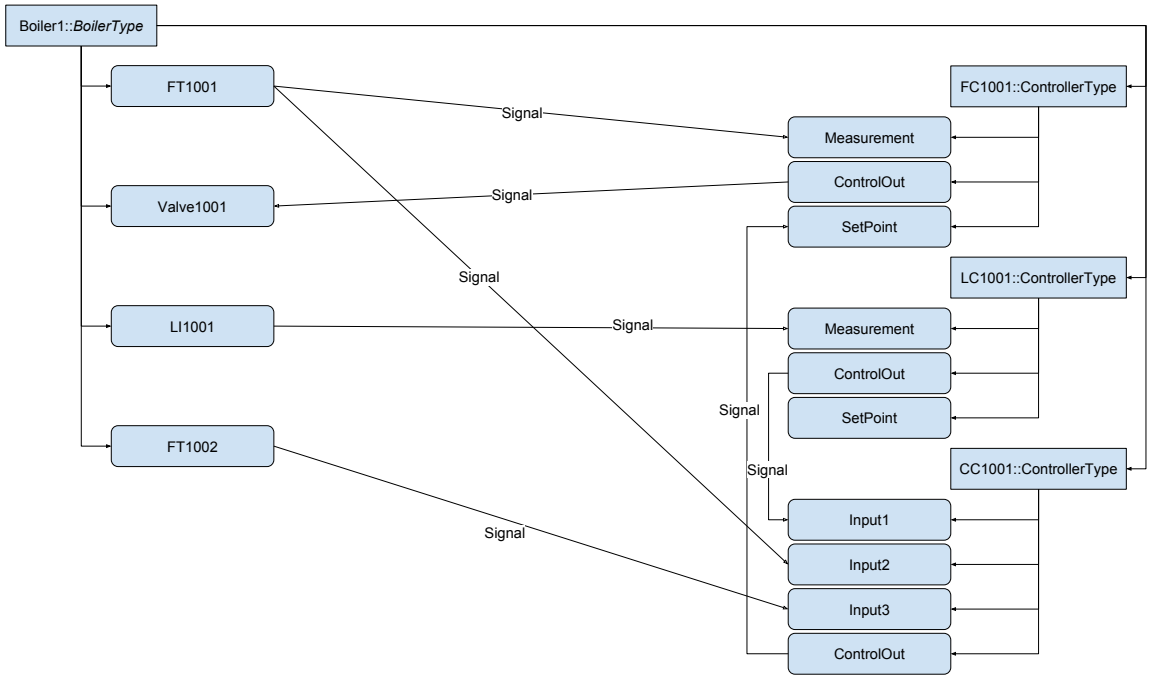

Figure 10: Transformed boiler address space

Looking at the differences between the two address spaces, we can see the need for four different rules: one to catch both DataItem nodes under the FTType, one to catch the DateItem under ValveType, one to catch the DateItem under LIType, and one to gather all three ControllerTypes and everything organized under them. Based on this information, the following rules were created:

1. Copy: [BoilerType]#1/[PipeType]/[FTType]#2/DataItem#3 = #1/#3(BrowseName=#2@BrowseName,DisplayName=#2@DisplayName)

2. Copy: [BoilerType]#1/[PipeType]/[ValveType]#2/DataItem#3 = #1/#3(BrowseName=#2@BrowseName,DisplayName=#2@DisplayName)

3. Copy: [BoilerType]#1/[DrumType]/[LIType]#2/DataItem#3 = #1/#3(BrowseName=#2@BrowseName,DisplayName=#2@DisplayName)

4. DeepCopy: [BoilerType]#1/[ControllerType]#2 = #1/#2

The first three are very similar, and all function as a regular Copy-rule. For the purpose of this case, the rules were designed to specify explicitly the entire path to test out the nuances between very similar rules. The first and third rules could

be combined into a more implicit version, by removing the middle two node type definitions, therefore producing a rule that matches both cases regardless of being a PipeType and FTType versus DrumType and LIType. The attribute values in the RHS are present to change the name of the resulting variable node to that of its immediate parent, be it a FTType or a LIType node.

The fourth rule will match with all three controller nodes: FC1001, LC1001 and CC1001. That rule is a deep copy, which means that it will also copy every node under the discovered node, using HasComponent references, creating a one-to-one clone of the source server from that part.

9.2 Reversed Boiler case

To prove the language is capable of reverting most, if not all, of the transformations back to the original form, the transformed boiler case from chapter 9.1 is taken as the source address space for this case. The reverse case is only done with a single instance of the transformed boiler as the source address space. The middle layers can be recreated through just initializing new node types on the target server, and all DataItem nodes can be recreated from the variable nodes in the source address space. Any types that were not copied in the first transformation process must be created from nothing.

Using the address spaces in figures 9 and 10 in reverse order compared to the previous case 9.1, the following four rules were constructed to recreate the original address space displayed in figure 9:

1. Copy: [BoilerType]#1/FT1001#2 = #1/[PipeType]Pipe1001/
   [FTType]FT1001/#2(BrowseName=DataItem,DisplayName=DataItem)

2. Copy: [BoilerType]#1/Valve1001#2 = #1/[PipeType]Pipe1001/
   [ValveType]Valve1001/#2(BrowseName=DataItem,DisplayName=DataItem)

3. Copy: [BoilerType]#1/LI1001#2 = #1/[DrumType]Drum1001/
   [LIType]LI1001/#2(BrowseName=DataItem,DisplayName=DataItem)

4. Copy: [BoilerType]#1/FT1002#2 = #1/[PipeType]Pipe1002/
   [FTType]FT1002/#2(BrowseName=DataItem,DisplayName=DataItem)

5. DeepCopy: [BoilerType]#1/[ControllerType]#2 = #1/#2

The first four rule's RHS all create new nodes, and new associated node types, if not already present in the target server. They have no reference information to let the engine know that they want to be explicitly newly created nodes with no association with the source server. The rules create the following types on the targer server: PipeType, DrumType, FTType, ValveType and LIType. The fifth rule creates a one-to-one copy of everything under every ControllerType -type nodes under the root BoilerType -type node, similar to how the rule 4 in the previous case in chapter 9.1 worked.

9.3 ISOBUS

As a final case, a real-life ISOBUS device server is used as a source address space for a transformation. This ISOBUS server was used in Elovaara's work [14], and was originally designed by Piirainen [27]. Piirainen designed the model to expose a selected group of the internal variables of a tractor and its implements. In his implementation, complex typing for objects, variables, and parameters were created, as displayed in figures 11 and 12.

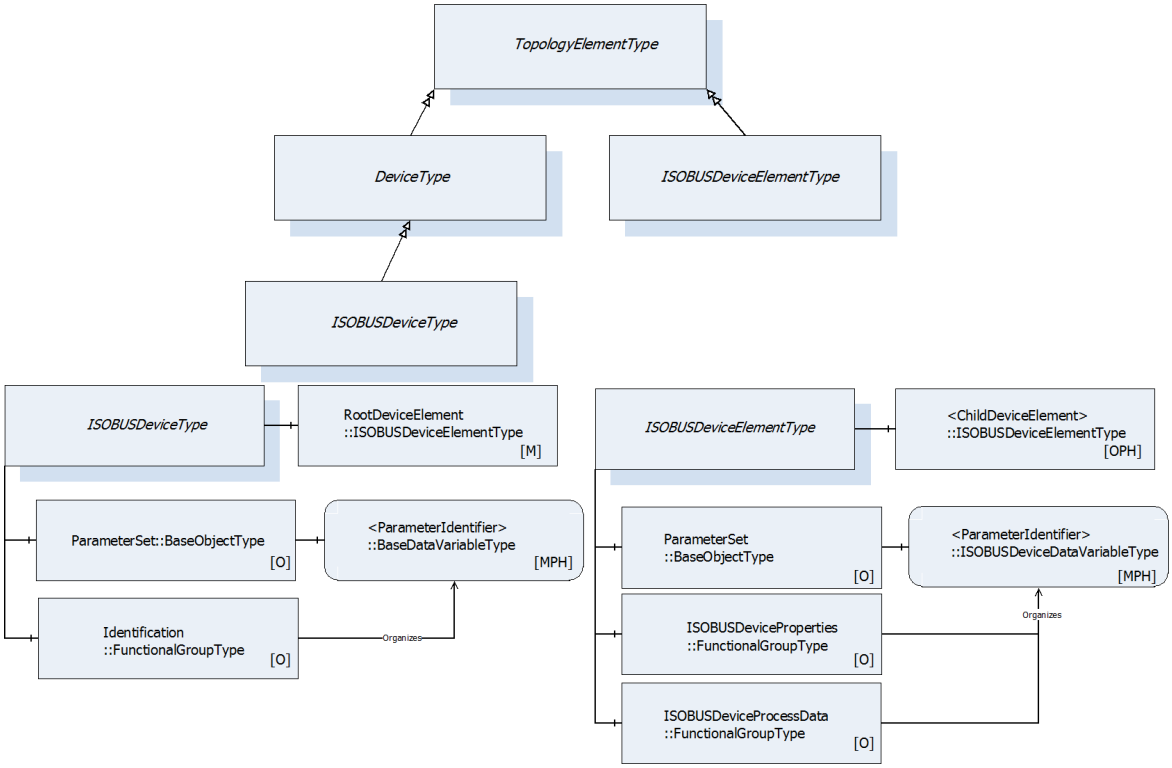

Figure 11: UA object types for the implements DDOP, adapted from Piirainen [27]. The arrow with two triangular heads represents a HasSubtype reference, and the arrow with the short perpendicular line as its head represents a HasComponent reference.

Figure 12: Variable types for ISOBUSDeviceElementType, adapted from Piirainen [27].

In the transformation process, we are interested in knowing the specific types available for the ISOBUS address space: ISOBUSDeviceType and ISOBUSDeviceElementType. While the knowledge of the parameters and variables available for each variable node is available, the current iteration of the language is not capable of representing the differences between the types, but it can still create restrictions based on the values of the parameters, so the names of the parameters are also usable information for creating rules. Fortunately, the model is cohesive enough to not require the use of parameters to discern between different nodes in the address space. The address space is also well-structured for the purposes of the transformation, as most, if not all, of the nodes to transform are placed under specifically named folders in the source address space.

The source address space consists of complex layering of the tractor inner structures, but each layer has conveniently organized all of the nodes to be transformed under an object called “ParameterSet”. The ParameterSet of each layer only contains the variables that are related to it directly, and none from any other ISOBUSDeviceType or ISOBUSDeviceElementType organized under it. Because the amount of layers is not a constant over different ISOBUS devices, each ruleset must contain

exactly the amount of rules that there are layers built into the address space model.

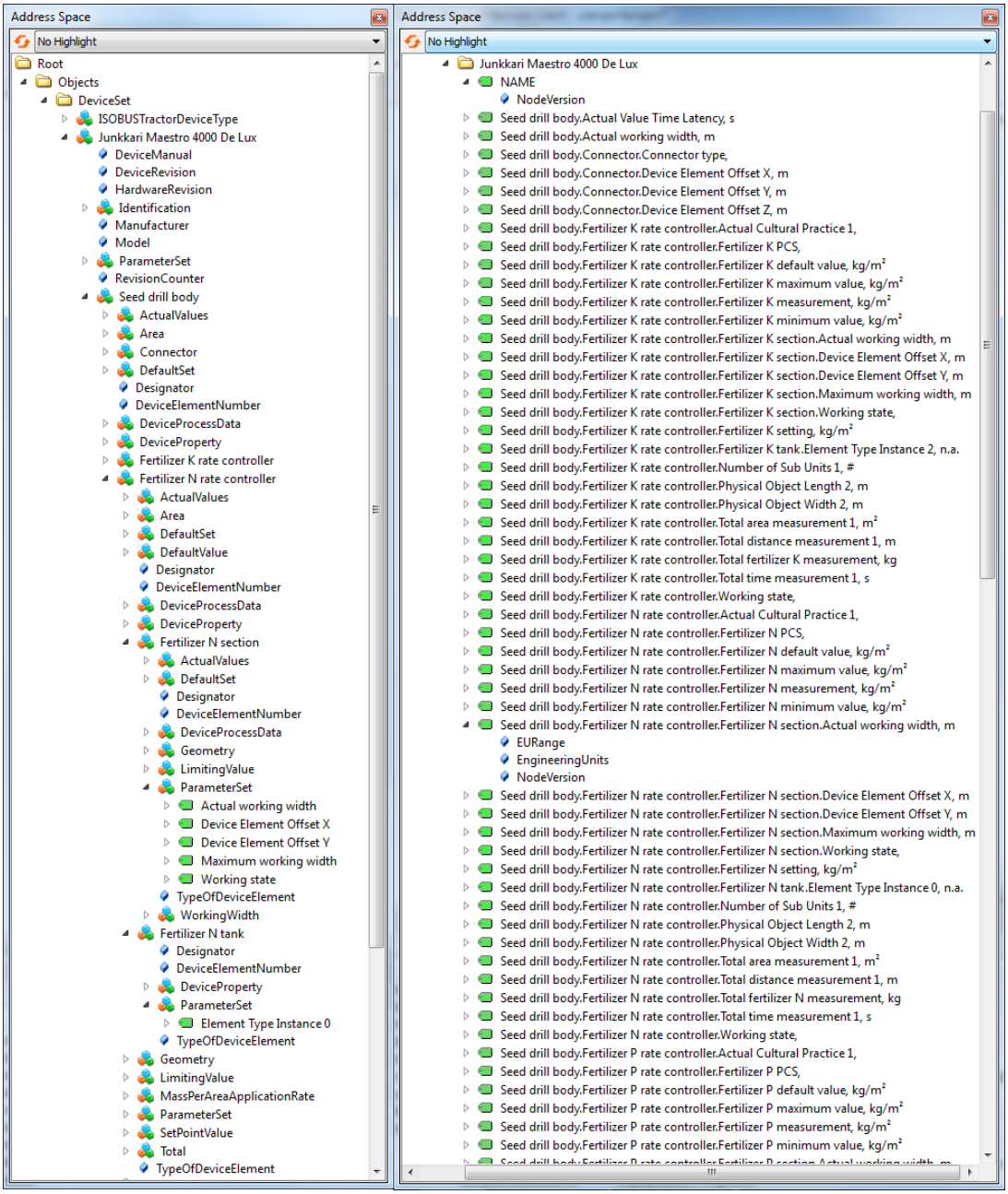

Figure 13: Original and aggregating address spaces for a Junkkari Maestro 4000 lux seed drill [14].

The following rule was created to perform the transformation displayed in the figure 13 for the first layer of ISOBUSDeviceElementTypes:

1. Copy: [ISOBUSDeviceType]#1/[ISOBUSDeviceElementType]#2/ParameterSet/#3=Tractors/#1/#3(DisplayName={#2@DisplayName}.{#3@DisplayName} {#3@EngineeringUnits})

In this rule, the DisplayName attribute of the resulting node is set to a combination of three other attribute and property values from the original node: its parent's DisplayName, its own DisplayName and the value of its parameter EngineeringUnits. For each layer deeper that you'd want to transform, an additional element needs to be added between the ISOBUSDeviceType and ISOBUSDeviceElementType, for example the following rule goes one level deeper than rule 1:

2. Copy: [ISOBUSDeviceType]#1/[ISOBUSDeviceElementType]#2/
   [ISOBUSDeviceElementType]#3/ParameterSet/#4=Tractors/#1/
   #4(DisplayName= {#2@DisplayName}.{#3@DisplayName}.{#4@DisplayName}
   {#4@EngineeringUnits})

In this rule, similarily to the previous one, the name is a combination of the DisplayNames of the node and its parent's, but also the grandparent's DisplayName. In any deeper layers represented as a rule, the naming only follows the immediate parent and grandparent of the current node, and any parents of the grandparent node are ignored. The combination of rule 1 and a necessary number of rule 2 iterations will create the required target address space also described in Figure 13.

10 Conclusions

The goal of this thesis was to design a rule language for performing OPC UA address space transformations. The research on model transformations yielded insight into how generic transformations should be done, most importantly the structure of a rule, and using the host language for the rule language could be reasonable for most transformation cases. In this case, however, the language was much easier to implement in a completely new language, as OPC UA itself is not ideal for representing partial nodes for the conditions of the LHS of the rule. This is because making the conditional part in OPC UA would not allow the representation of partial or generic values in the elements. Partial values in this sense means elements, where only one attribute or type would be necessary, and would require the generation of otherwise empty elements, and generic values means that any value at all would satisfy the condition.

The first step in the process was to discover the requirements for such a language. This was done based on initial experimentation with OPC UA and the test cases described in chapter 9. The requirements defined a rule language which would be capable of performing transformations for almost any address space into other address spaces, as long as the address space followed the OPC UA specification and the data nodes were a subset of the data nodes present in the source address space. Because there were no previous scenarios of OPC UA address space, the stakeholders that would be using the transformation language can not be accurately described, only guessed at. Some use cases and users could be discerned through similarities with other transformation languages, such as XSLT, and the requirements were adjusted based on this knowledge. The most likely users would be the developers of a software that would be using the devices, or the configurators of a similar software, but one that could take any properly configured address space as a source. With these requirements in mind, a prototype rule language could be developed and used to transform the aforementioned cases.

While designing the rule language, many other similar languages were used as a starting point, including but not limited to XSLT [10], XPath [11] and regular expressions [15]. Due to time and manpower constraints, the designed rule language did not include all of the possible features and requirements found in the requirements part of the thesis, but all of the most important ones that made the language to be capable of functioning in most cases were. This means that the current version of the language is not capable of handling every single corner case and every imaginable transformation a user might come up with, but it can still transform at least multiple instances of the same metamodel at the same time, and combine multiple sources through separate transformation processes into one aggregating server address space, and all the single features covered by the test cases. The implementation of the rule language and transformation engine were created using a pre-existing aggregating server platform designed and implemented by Elovaara [14].

To test the implemented rule language against the requiements, three cases were designed that would test at least the most common scenarios the language could be used for. The transformation process in the test cases was done through planning the

rules beforehand by knowing the differences between the source and target address space models. Both address space models contained all of the information relevant to the transformation process, most importantly the types and names of each node. The comparison between the models allowed the generation of rules that created the connections between each value node in the models. Then, the transformation process would be run, and the results were compared to the previously designed target address space. Differences between the models revealed any issues in the rule language, which were then fixed through iteration.

The implemented rule language prototype was capable of performing the transformation required by the test cases successfully. Each transformation resulted in generating an address space that contained all of the nodes that matched the LHS of each rule generated properly based on the RHS of those rules. The data updated from the source address space succeeded in updating the aggregating nodes, and subscriptions made on the aggregating server updated along with the data changes in the source address space. Comparing the results of the test case transformations to the design and requirements of the rule language, it can be deducted that the designed rule language would be capable of performing most types of single source address space transformations that could be designed by the stakeholders. However, after further examination of the developed prototype, some improvement suggestions were found. Those are described in the Further development chapter 10.1, to improve the features of the rule language, and expand the possibilities of what such a transformation language could be capable of.

10.1 Further development

The rule language currently represents a single rule with a single line of text. This single line restriction could create issues with very complex pattern matching, but was enough to perform the transformations described in the test cases in chapter 9. Also, the more complex the rule would become, the harder it would be to read if it only used a single line of text. The next step in the development would be to modify the language to use multi-line descriptions instead. This would enable many new features compared to the current system, such as reference types and directions, binary operations such as AND, OR, and NOT, and allowing complex cross-referencing and comparisons between nodes themselves during the rule writing process. It would also release the restricted characters back into the names and type names of the nodes, even though those characters are very rarely used in them in any case.

An efficient way to create multi-line rules would be to use an existing markup language, such as XML or JSON. They would allow very complex structures to be built, including just defining a reference that does not point to a node at all. In programming terms, the source or target node of the reference could be NULL. An example of a possible JSON-based language LHS of a rule is in figure 14. It is the same rule that was presented in chapter 9.1 rule 1.

```
{
        RuleType: “Copy”,
        Type: “BoilerType”,
        ReferenceName: “1”,
        Reference:
        {
                Type: “HasComponent”,
                TargetNode:
                {
                        Type: “PipeType”,
                        Reference:
                        {
                                Type: “HasComponent”,
                                TargetNode:
                                {
                                        Type: “FTType”,
                                        ReferenceName: “2”,
                                        Reference:
                                        {
                                                Type: “HasComponent”,
                                                TargetNode:
                                                {
                                                        DisplayName: “DataItem”,
                                                        ReferenceName: “3”
                                                }
                                        }
                                }
                        }
                }
        }
}
```

Figure 14: Example JSON rule LHS.

An alternative way to release the restricted characters back to the node names and type names would be to introduce an escape character, similar to how regular expressions and most other languages with an escape character handle special characters: the '\'-character. When used immediately before a special character, the interpreter ignores the character as a special character and assumes it to be a part of the text input in the rule.

As described earlier, the rule language could also include some or all of the following features.

1. Each parameter in the rule, be it a type, reference, name or a part of a name or value, could be preceded by an operator, such as AND, OR, or NOT. Enabling the use of OR would allow multiple almost identical rules to be combined into one, and the user of NOT would let the user exclude some specific case without having to design multiple rules to cover every other case instead.

2. Reference types should be included in the rule as a parameter. Currently the language only considers the HasComponent and Organizes references, and while those two references usually cover all of the nodes in an industry environment A, it might not be the case in every possible scenario. It would also enable complex conditions through the usage of custom references present in the address spaces.

3. A single rule could take nodes from two different address spaces. This would only be relevant for the RHS of the rule, where the target address space could be built using nodes and types from two different address spaces, without the

need of creating these types again completely, but instead the types could be copied directly from the two or more separate source address spaces. This would be useful for creating a coherent combination address space of similar sources, for example two very similar devices and their address spaces created by two different companies.

References

[1] MatrikonOPC. http://www.matrikonopc.com/. Accessed: 26.11.2015.

[2] OPC Foundation. https://opcfoundation.org/. Accessed: 26.11.2015.

[3] Prosys OPC. https://www.prosysopc.com/. Accessed: 26.11.2015.

[4] Prosys OPC UA Historian. https://www.prosysopc.com/products/opc-ua-historian/. Accessed: 26.11.2015.

[5] Softing. http://industrial.softing.com/en/. Accessed: 26.11.2015.

[6] Unified Automation. https://www.unified-automation.com/. Accessed: 26.11.2015.

[7] Irina Astrova, Nahum Korda, and Ahto Kalja. Storing OWL ontologies in SQL relational databases. In *INTERNATIONAL JOURNAL OF ELECTRICAL, COMPUTER, AND SYSTEMS ENGINEERING*, volume 1, 2007.

[8] G. Candido, F. Jammes, J. B. de Oliveira, and A. W. Colombo. SOA at device level in the industrial domain: Assessment of OPC UA and DPWS specifications. In *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, pages 598–603, July 2010.

[9] Namyoun Choi, Il-Yeol Song, and Hyoil Han. A survey on ontology mapping. *SIGMOD Rec.*, 35(3):34–41, September 2006.

[10] James Clark. XSL Transformations (XSLT) version 1.0. World Wide Web Consortium, Recommendation REC-xslt-19991116, November 1999.

[11] James Clark. XML Path Language (XPath) version 1.0. World Wide Web Consortium, Recommendation REC-xpath-19991116, September 2015.

[12] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA*, Anaheim, CA, USA, 2003.

[13] Dragan Djuric, Dragan Gasevic, and Vladan Devedzic. Ontology modeling and MDA. *Journal of Object Technology*, 4(1):109–128, 2005.

[14] Joona Elovaara. Aggregating OPC UA server for remote access to agricultural work machines. Master's thesis, Aalto University School of Electrical Engineering, 2015.

[15] Daniela Florescu, Alon Levy, and Dan Suciu. Query containment for conjunctive queries with regular expressions. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, pages 139–148, New York, NY, USA, 1998. ACM.

[16] Robert B. France, Sudipto Ghosh, Eunjee Song, and Dae-Kyoo Kim. A metamodeling approach to pattern-based model refactoring. *IEEE Software*, 20(5):52–58, 2003.

[17] Lars Marius Garshol. BNF and EBNF: What are they and how do they work?, 08 2008. Accessed 08,03.2016.

[18] Daniel Grosmann, Markus Bregulla, Suprateek Banerjee, Dirk Schulz, and Roland Braun. OPC UA server aggregation - the foundation for an internet of portals. In Antoni Grau and Herminio Martínez, editors, *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation, ETFA 2014, Barcelona, Spain, September 16-19, 2014*, pages 1–6. IEEE, 2014.

[19] Joann T. Hackos. What is an information model & why do you need one? *The Gilbane Report*, 10(1), February 2002.

[20] Alon Halevy and et al. Data integration: The teenage years. In *IN VLDB*, pages 9–16, 2006.

[21] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Q.*, 28(1):75–105, March 2004.

[22] Georg Hinkel. An approach to maintainable model transformations using an internal DSL. Master's thesis, Karlsruhe Institute of Technology, October 2013.

[23] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph, editors. *OWL 2 Web Ontology Language: Primer*. W3C Recommendation, 27 October 2009. Available at http://www.w3.org/TR/owl2-primer/.

[24] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. *OPC Unified Architecture*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[25] OPC Foundation. OPC UA specifcation part 3: Address space model. Technical report, OPC Foundation, 2012. OPC UA Specifcation Release 1.02.

[26] OPC Foundation. OPC UA specifcation part 8: Data access. Technical report, OPC Foundation, 2012. OPC UA Specifcation Release 1.02.

[27] Pyry Piirainen. OPC UA based remote access to agricultural field machines. Master's thesis, Aalto University School of Electrical Engineering, 2014.

[28] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2), February 2006.

[29] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45, September 2003.

[30] Dániel Varró and András Pataricza. Generic and meta-transformations for model transformation engineering. In Ana Moreira Thomas Baar, editor, *UML 2004 - The Unified Modeling Language*, volume 3273 of *Lecture Notes in Computer Science*, pages 290–304. Springer-Verlag, October 2004.

[31] Laurent Wouters and Marie-Pierre Gervais. Ontology transformations. In *16th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2012, Beijing, China, September 10-14, 2012*, pages 71–80, 2012.

A Example of a real-life address space definition

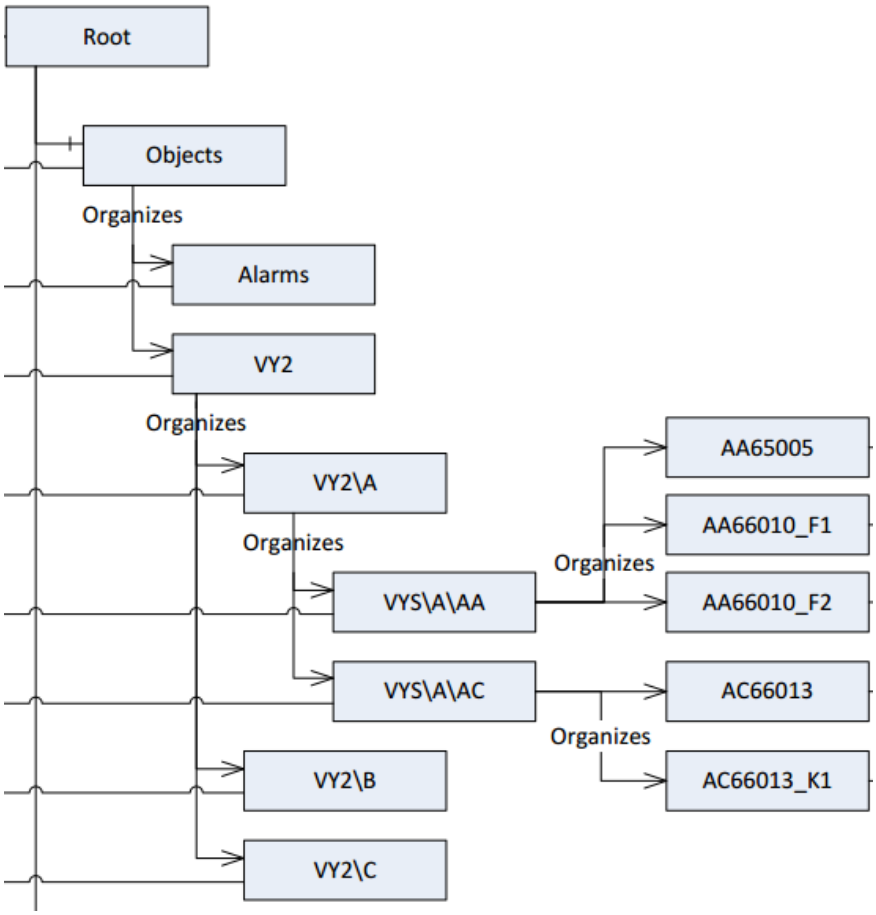

Figure A1: An example of a real-life address space built using type definitions. Notice how all of the nodes can be accessed through Organizes references inside of the Objects folder.

B Example of a real-life address space type definition

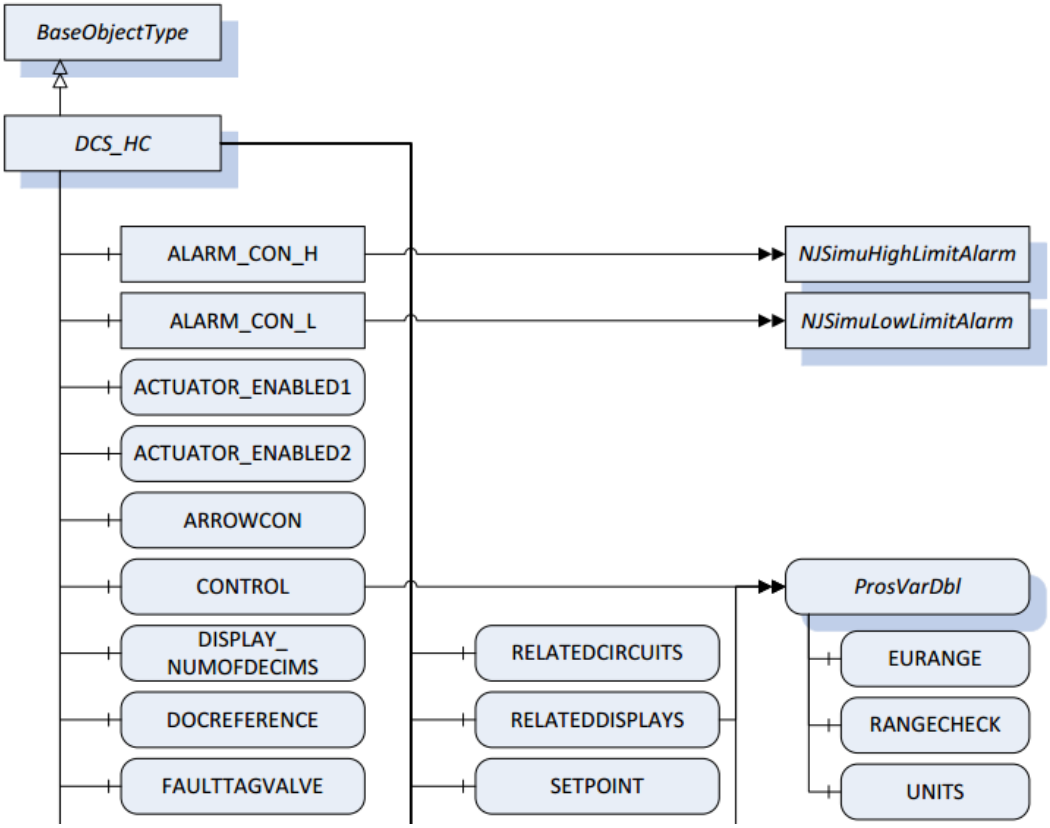

Figure B1: A type inherited from the BaseObjectType with parameter and reference definitions and connections.