Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Implementing an Enumerative Semantic Differencing Operator for OPC UA

Patrick Spaney

Course of Study:	Informatik
Examiner:	Prof. Dr.-Ing. Steffen Becker
Supervisor:	Jun.-Prof. Dr. rer. nat. habil. Andreas Wortmann
Commenced:	December 22, 2021
Completed:	June 22, 2022

Abstract

The OPC Unified Architecture (OPC UA) is a widely adopted set of communication standards for industrial automation that provides its own extensible data model. To understand the evolution of models over time and the differences between model versions, differencing is an essential tool.
Differencing is commonly done syntactically, e.g., consisting of add and delete operations for model elements that transform one version into the other. In contrast to the usual syntactic approach, differencing can also be done on a semantic level by providing a model or an enumeration of instances, so-called witnesses, that can be derived from one model but not from another.
OPC UA's semantics, i.e., the rules for instantiation of its types, are rather complicated. This can make the precise semantic differences between model versions difficult to see. Furthermore, OPC UA information models may often be designed by domain experts of various engineering disciplines with little prior modeling experience.
To tackle this problem, we introduce *uadiff*, an enumerative semantic differencing operator for OPC UA models. Moreover, *uadiff* uses a complete interpretation, i.e., references are only present on instances if they are present in the type definition. We aim to simplify comparisons of model versions and thus model development, reduce redundant models and through our stricter instantiation rules improve the overall model quality.
To our knowledge, no semantic differencing operator exists for OPC UA models. However, such operators have been described and implemented for other modeling languages, such as UML class diagrams (CDs). We present an ATL transformation from OPC UA to CD, which, in composition with a slightly modified CD semantic differencing operator, forms *uadiff*. The combination of these components lets *uadiff* preserve many semantic intricacies that are not present in UML.
The *uadiff* operator is implemented as a Java demonstrator application executing the transformation and applying the modified CD differencing solution to the results. We demonstrate the operator and its implementation with a simple example and evaluate the performance on models relevant in practice.
We conclude the thesis with a discussion on the applicability and limitations of our approach, as well as an outlook on possible future research.

Kurzfassung

Die OPC Unified Architecture (OPC UA) ist eine verbreitete Sammlung von Kommunikationsstandards für industrielle Automatisierung, welche ihr eigenes erweiterbares Datenmodell bereitstellt. Differencing ist ein essentielles Werkzeug um die zeitliche Evolution von Modellen nachzuvollziehen.
Gewöhnlich handelt es sich um syntaktisches Differencing, welches beispielsweise aus add und delete Operationen besteht, die eine Version des Modells in die andere überführen. Im Gegensatz zum üblichen syntaktischen Ansatz kann Differencing auch auf semantischer Ebene betrieben werden, indem man ein Modell oder eine Aufzählung der Instanzen erzeugt, welche aus einem Modell instanziiert werden können, nicht aber aus dem anderen.
OPC UAs Semantik, also die Regeln für die Instanziierung der Typen, sind verhältnismäßig kompliziert. Dadurch kann es schwierig sein die exakte semantische Differenz zwischen zwei Versionen eine Modells zu erkennen. Darüber hinaus kommt es vor, dass OPC UA Informationsmodelle von Domänenexperten anderer Ingenieursdisziplinen erstellt werden, welche wenig Erfahrung mit Modellierung haben.
Um dieses Problem anzugehen stellen wir *uadiff* vor, einen enumerativen semantic Differencing Operator für OPC UA Modelle. Darüber hinaus nutzt *uadiff* eine vollständige Interpretation von OPC UA Modellen, d.h. Referenzen von Instanzen sind nur dann vorhanden, wenn der jeweilige Typ diese auch definiert. Wir streben damit an, Vergleiche verschiedener Versionen von Modellen zu vereinfachen sowie eine Reduktion redundanter Modelle und eine allgemeine Verbesserung der Qualität von OPC UA Modellen zu erreichen.
Unserem besten Wissen nach existiert bislang kein semantic Differencing Operator für OPC UA Modelle. Jedoch wurden solche Operatoren für andere Modellierungssprachen beschrieben und implementiert, so z.B. für UML Klassendiagramme (CDs). Wir stellen eine ATL Transformation vor, welche zusammen mit einem leicht modifizierten Differencing Operator für CDs *uadiff* bildet. Die Kombination beider Komponenten erlaubt es *uadiff*, viele semantische Feinheiten der OPC UA beizubehalten, welche so nicht in UML vorkommen.
Der *uadiff* Operator wird als eine Java Demonstrator-Anwendung implementiert, welche die Transformation ausführt und die modifizierte OPC UA Differencing Lösung auf die Ergebnisse anwendet. Wir führen eine Demonstration des Operators und seiner Implementierung an einem einfachen Beispiel durch und evaluieren die Performanz für praxisrelevante Modelle.
Abschließend erörtern wir die praktische Anwendbarkeit sowie die Einschränkungen unseres Ansatzes und geben einen Ausblick auf mögliche zukünftige Forschung.

Contents

List of Figures

List of Tables

Listings

Acronyms

API application programming interface. 40

AST abstract syntax tree. 14, 19, 34, 37, 40, 41

ATL Atlas Transformation Language. iii, v, xiii, 2, 6, 7, 8, 19, 29, 31, 32, 33, 34, 35, 38, 39, 40, 41, 43, 44, 45, 49, 50, 58, 59, 60

CD class diagram. iii, v, 1, 2, 9, 14, 15, 17, 19, 20, 21, 23, 26, 27, 29, 32, 34, 39, 40, 43, 46, 47, 49, 50, 51

CD4A Class Diagrams for Analysis. 6, 14, 16, 17, 19, 20, 24, 29, 34, 37, 40, 41, 49, 51

CD4C Class Diagrams for Code. 14

CLI command-line interface. 40

DSL domain specific language. 14

EMF Eclipse Modeling Framework. 6, 8, 14, 18, 40, 41

EMFTVM EMF Transformation Virtual Machine. 6, 7, 34, 40, 41, 44, 46

MC MontiCore. 14, 18, 40

MDE model-driven engineering. 1, 6

MOF Meta Object Facility. 6

OCL Object Constraint Language. 7

OOP object-oriented programming. 9

OPC Open Platform Communications. 1

OPC UA OPC Unified Architecture. iii, v, ix, xi, xiii, 1, 2, 5, 6, 7, 8, 9, 10, 12, 13, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 29, 30, 38, 39, 40, 41, 43, 44, 45, 46, 49, 50, 51, 65, 66

SAT Boolean satisfiability problem. 5, 29

UML Unified Modeling Language. iii, v, xi, 1, 2, 6, 9, 14, 16, 17, 18, 20, 22, 24, 34, 45, 49, 50

VM virtual machine. 40

XML Extensible Markup Language. 19, 32, 41

XSD XML Schema Definition. 6, 8, 19, 40, 41

1 Introduction

The OPC Unified Architecture (OPC UA) is a set of standards for communication in industrial automation announced by the OPC foundation in 2006 [SB13]. It was designed as successor for Open Platform Communications (OPC),which included several interface specifications.
OPC UA combines the different functionalities of these interfaces while providing platform independence and security. Furthermore, an extensible information model is defined within the specification using a metamodel and a base information model, which is an instance of the metamodel [LM06; OPC20; OPC21b]. Vendors can construct new information models by creating their own instances of the OPC UA metamodel.

In model-driven engineering (MDE) model differencing is an important tool that allows to better understand the evolution of models over different versions. It is commonly used to compare two models within a common version history. Most differencing operators are purely syntactical, where the difference usually consists of the creation and deletion of model elements. While this kind of operator is useful, it does not take semantics, i.e., what instances can actually be derived from the models, into account and thus might find semantically similar models to be very different from each other or vice versa.
To address this problem a generic semantic differencing operator was proposed by Maoz et al. [MRR10]. The abstract operator takes two models as input and outputs so-called diff witnesses, which are instances of the first model but not of the second model. Using this property, several relationships between model versions can be checked, such as both models being semantically equivalent to each other or one model being a refinement of the other. The difference can also be summarized by defining distinct equivalence classes of diff witnesses and computing a representative for each of the respective classes [MRR12].
There are existing concrete instances of the operator, namely *CDDiff* for class diagrams (CDs) [MRR11b] and *ADDiff* for activity diagrams [MRR11a]. In the course of this work we will focus on the former, as the metamodels for OPC UA and Unified Modeling Language (UML) CDs share many similarities.

Instantiation rules for OPC UA, as defined in [OPC21b], are quite complicated due to several factors, most prominently different levels of abstraction are not as clearly separated as in, e.g. CDs.
OPC UA models contain so-called *TypeDefinitionNodes*, instances of the metamodel elements *ObjectType* and *VariableType*. The *TypeDefinitionNodes*' *Instances*, as defined by a *HasTypeDefinition Reference*, are actually instances of other metamodel objects, *Object* and *Variable*. Additionally, these *Instances* can be part of a type definition if directly or indirectly referenced by a *TypeDefinitionNode* and these *References* can be overridden in subtypes of the *TypeDefinitionNode*. Furthermore, *Instances* can contain almost arbitrary *References* regardless of their type's *References*. These aspects make it difficult to understand OPC UA semantics and to determine the difference between model versions on a semantic level.

To address this problem we first propose a *complete* interpretation, similar to the complete interpretation of CDs in [MRR11b; Rum11], with restricted instantiation rules for OPC UA.
Such an interpretation makes the models easier to comprehend and enables the use of semantic differencing in a meaningful way. Based on this interpretation we present semantics for OPC UA models, mapping valid OPC UA models to UML object models. Using this mapping we can define an OPC UA specific instance of the semantic differencing operator described in [MRR10].

The semantic mapping is constructed using an Atlas Transformation Language (ATL) transformation [JABK08] from OPC UA to UML and a slightly modified version of *cddiff*'s semantic mapping. We examine the rules for instantiation of OPC UA types in detail and construct the mapping in such a way, that OPC UA's special semantics are preserved. As a proof of concept we implemented the operator as a Java application. It transforms subsets of both input models by executing the ATL transformation and passes the output CDs to the *cddiff* implementation [MRR11b].

We demonstrate how key parts of OPC UA semantics are reflected by our implemented operator using a simple model of our own creation. Additionally, we evaluate the performance of the demonstrator on several types from OPC UA models used in practice, measuring the execution time of the ATL transformation and counting the produced CD elements. The latter is an indicator of the expected *cddiff* performance [MRR11b]. The results of this performance evaluation demonstrate the basic viability of our implementation approach for use in practice.

We discuss the achieved results, including benefits and limitations. The limitations include some inherent difficulties of applying semantic differencing to OPC UA in a meaningful way and some weaknesses of our implementation approach. Finally we also discuss possible future research, including improved presentation of witnesses and extending the implementation or using a different approach.

The contribution of this thesis is a semantic differencing operator for OPC UA, simplifying the analysis and understanding of OPC UA model evolutions through comparison of two model versions. This includes a refined interpretation of OPC UA models for an approach to OPC UA modeling with full-fledged types that concisely define the form of their instances instead of just defining a minimal set of required references.

Thesis Structure

The remainder of the thesis is divided into the following chapters:

Chapter 2 – Foundations and Related Work: Here, we provide foundations for concepts and technologies used in the thesis. Additionally, we discuss related work.

Chapter 3 – UADiff Operator: In this chapter we conceptually describe the structure of our operator and the involved mappings.

Chapter 4 – Implementation: Following the conceptual part, we move on to the concrete implementation and, by means of ATL transformation, definition of our operator.

Chapter 5 – Evaluation: Afterwards we demonstrate the correctness for key aspects of our implementation on a simple example and evaluate the performance on several OPC UA models in practice.

Chapter 6 – Conclusion We conclude our thesis with a brief summary, a discussion of benefits and limitations and an outlook on possible future research.

2 Foundations and Related Work

In this chapter we give introductions to the concepts and technologies used in the context of this work. We also discuss some related work and the reasoning why we chose this approach to semantic differencing for OPC UA.

2.1 Foundations

Two important concepts used in this thesis are model-to-model transformations and differencing, specifically semantic differencing. Therefore we give a brief introduction to both concepts, as well as to specific frameworks, tools and languages we used in the process of implementing our operator. We cover the type model of OPC UA information models [OPC21b] extensively, as a detailed understanding of the way types are instantiated in OPC UA is essential to this work. For other software and languages used in the context of this work we do not require in-depth knowledge and will therefore be more brief in the respective sections.

2.1.1 Alloy

Alloy is a specification language using first order logic constructs [Jac19] that is analyzed via its corresponding tool, the Alloy Analyzer [JSS00]. The Alloy Analyzer translates Alloy modules to Boolean formulas. This can be accomplished by limitation of the module to a finite scope. The generated formulas are then solved by standard SAT solvers and the solutions are translated back. We will not go into details regarding Alloy since we only did some minor modifications to the Alloy module generated by the *cddiff* implementation and covering Alloy extensively would exceed the scope of this work. To help the reader understand the small modifications to the existing Alloy generator in *cddiff*, we shall briefly cover the relevant Alloy constructs:

sig Signatures are used to define sets. The declaration of a signature can also include relations. Together these sets and relations are the underlying data structures of Alloy. Elements of these sets are called atoms.

fun Functions in Alloy are relations that map their given input to some output. The output is itself a relation.

pred Predicates are similar to functions, but they return either true or false. With the *run* command instances of the model for which the predicate evaluates to true are searched.

fact Facts are used to constrain models. They are similar to predicates, but they have no input and are assumed to be always true. When searching for model instances with a *run* or *check* command, any model instance for which the predicate evaluates to false is discarded, which means such instances are not valid counter examples when running a *check* command.

2.1.2 Model-to-Model Transformations

Model-to-model transformations are a core concept in MDE, mapping source models to target models and executing the mapping in an automated fashion [Béz05; WHR13]. The transformations are often described in a model transformation language, such as ATL, that requires a target metamodel the produced target models conform to, while the source models conform to a source metamodel [JABK08]. The model transformations in this work are purely exogenous horizontal model transformations, following the taxonomy proposed by Mens and Van Gorp [MV06], meaning the model will be transformed from one modeling language to another and remain at the same level of abstraction.

EMF

The Eclipse Modeling Framework (EMF) is "*a [modeling] framework and code generation facility*"[SBMP11, p. 14]. Models handled by the EMF must conform to the Ecore Metamodel, which is similar to the MOF [Obj06], the metamodel for UML.

Models can be created or edited directly in Eclipse or imported from a variety of sources, including XML Schema Definitions (XSDs). Once a model is available in the EMF registry, Java code can be generated from it. EMF-generated code provides classes corresponding to the model elements and additional facilities needed to process the models. The models are structured in *EPackages*, which are registered at an *EPackageRegistry*. Each package registers all contained classes, interfaces, and any relationships between them. Additionally, an *EPackage* provides an *EFactory* for creating instances of each class. Not all of these facilities are strictly necessary for every use case, since metamodels can also be created dynamically at runtime. However, they are needed for the OPC UA and CD4A metamodels in the context of this work.

EMF also enables the use of various tools on EMF models. One such example are model-to-model transformations, which can be described in a model transformation language, such as ATL [WTCJ11].

ATL

The Atlas Transformation Language (ATL) is "*a domain-specific language for specifying model-to-model transformations*" [JABK08].
Every ATL transformation needs a source and a target metamodel, both conforming to a common meta-metamodel, which for transformations between EMF models is Ecore. ATL modules themselves are instances of the ATL metamodel, which also conforms to the mentioned meta-metamodel. ATL uses both declarative and imperative concepts, but the use of declarative over imperative code is heavily encouraged. In this work we use ATL with the EMF Transformation Virtual Machine (EMFTVM) [WTCJ11], therefore some details might differ from standard ATL. The ATL code is compiled to EMFTVM bytecode and then executed by the EMFTVM.

In ATL every transformation is defined by a single module, which contains rules, helpers and a mandatory header with general information.

The rules are divided into several categories:

```
module SimpleUA2CD;
create OUT : CD4A from IN : UA;

rule ObjectType2Class {
  from
    --source pattern, matches all abstract ObjectTypes
    objectType : UA!ObjectType (
      not objectType.isAbstract
    )
  using{
    className : String = objectType.browseName; --define a local variable
  }
  to
    --target pattern, creates a Class with the same name as the ObjectType
    class : CD4A!ASTCDClass (
      Name <- className;
    )
  do {
    class.Modifier <- thisModule.createUmlModifier(objectType); --imperative code
  }
}
```

Listing 2.1: Simple ATL module with a single matched rule. Contains a local variable section and an imperative section in addition to the source and target patterns.
The code is simplified and does not correspond to any transformation used in this work.

Matched Rules always have a source pattern and a target pattern. A matched rule is executed for each source element that matches the source pattern, thus creating the target elements specified in the target pattern. The source pattern must consist at least of a source type from the source metamodel and may contain additional conditions as a Boolean expression. Target model elements created by a matched rule or unique lazy rule can be retrieved via the respective source element using EMFTVM's tracing mechanism.

(Unique) Lazy Rules are special matched rules. They are only invoked when called by another rule. Lazy rules create new target elements each time they are called while unique lazy rules always return the same target elements for the same source elements.

Called Rules do not have a source pattern but may have a target pattern. They are called from other rules to create target elements in an imperative way and may have parameters. The special entrypoint and endpoint called rules constitute an exception to this. The entrypoint rule, if present, is called after initialization but before the matching phase for the matched rules. Analogously the endpoint rule, if present, is called after the matching phase.

Every rule may also have a local variables section and an imperative section. The structure of a matched rule containing all possible sections is shown in listing 2.1.
The data types in ATL, including collections, are based on those in OCL. Additionally, metamodel types can be used with some restrictions, depending on the type being part of a source or a target metamodel.

Helpers are functions that can be defined in the context of a type or the module. They return a single value and can take several input parameters. A simple helper in the context of a source metamodel type is shown in 2.2. The use of helpers reduces code-redundancy which is why helpers can be collected in libraries for reuse in other transformations. A byproduct of this work is a library with a number of OPC UA specific helpers.

```
--This helper returns the target node of a reference
helper context UA!Reference def : getTargetNode() : UA!UANode =
    let targetId : String =
        self.value
    in
        thisModule.getNodeById(targetId)
    ;
```

Listing 2.2: Simple ATL helper for transformations with OPC UA source models.

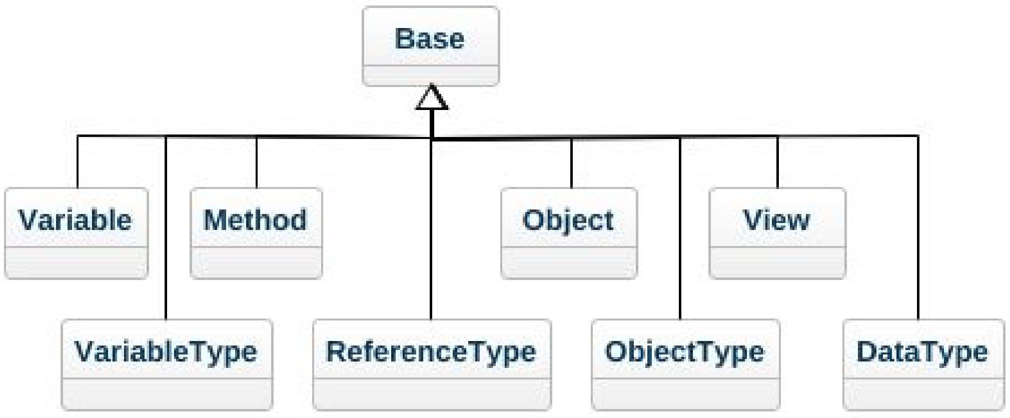


Figure 2.1: OPC UA *NodeClasses* [OPC21b]

2.1.3 OPC UA

The OPC UA metamodel is given in the OPC UA Address Space Model [OPC21b], which defines the basic elements as well as rules for well-formed models and instantiation of types. In order to define an appropriate semantic differencing operator, we need to understand precisely how an instance is created from its type in OPC UA. To this end, we give an introduction to the most important concepts in [OPC21b]. Note that the XSD file, from which we generate the EMF-conforming OPC UA metamodel for *uadiff* does not include many of the well-formedness rules from [OPC21b] and we assume only correct models to be used as input for our operator.

Models in OPC UA consist of *Nodes* connected by *References* contained in the *Nodes*. *Nodes* can be of different *NodeClasses* (see figure 2.1), which are defined in the metamodel. Each of these *NodeClasses* has a set of *Attributes* associated with its instances that provide additional metadata about the *Nodes*. The set of *Attributes* is defined by the specification for each *NodeClass* and cannot be extended. Every *Node* can be uniquely identified by its *NodeId Attribute*. A *Node* also has a *DisplayName* and a *BrowseName*.
For *References* the containing *Node* is called the *SourceNode* and the referenced *Node* is called the *TargetNode*.
The latter is important for instantiation, but both names may not be unique within a model.

The *NodeClasses* as defined by the specification are the following:

ObjectType Represents a type of *Object* and models common properties for all *Objects* of the type. OPC UA, however, does not separate these two levels of abstraction. The metamodel element *ObjectType* corresponds closely to the metamodel element *Class* in UML and *ObjectType* instances therefore correspond to classes.

Object A specific instance of an *ObjectType*. It represents a (real-world) asset by organizing the respective data and offering *Methods* for interaction. While each *Object* is an instance of the metamodel element *Object*, it is also in a type-instance relationship with an *ObjectType*, similar to classes and objects in object-oriented programming.

VariableType The types for *Variables*, although again residing on the same level of abstraction. Similarly to *ObjectType*, the metamodel element is comparable to *Class* in UML and the instances to classes.

Variable A specific instance of a *VariableType*. A *Variable* can be a *DataVariable* or a *Property*. A *DataVariable* represents the contents of an *Object* either directly or as a complex *DataVariable*, which references further *Variables*. A *Property*, on the other hand, provides characteristics of an *Object* or *Variable*, such as the engineering unit for a *DataVariable*. Analogously to *Objects*, each *Variable* is an instance of the metamodel element *Variable*, but it is also in a type-instance relationship with an *VariableType*, similar to classes and objects in object-oriented programming.

ReferenceType Type definitions for *References*. The OPC UA specification defines a set of standard *ReferenceTypes* for different purposes.

DataType As the name suggests, *DataType Nodes* are used to define data types. They describe the structure of the *Value Attribute* for *Variables* and are therefore referenced by *VariableTypes* and their instance *Variables*.

Method Similar to methods in object-oriented programming (OOP), *Methods* in OPC UA represent functions in the scope of an *Object* or *ObjectType*. They may affect the state of the respective *Object*. We do not go into further detail here because *Methods* will be ignored by our operator.

View Defines a subset of the *Nodes* in the potentially very large *AddressSpace* in order to present only the relevant information for a specific context. While not directly relevant for the differencing operator, they could be a way to define the input subset of an *AddressSpace* directly from within OPC UA.

OPC UA also defines its own graphical notation. The notation for *Nodes* and *References* used in this work can be found in tables C.1 and C.2.
While OPC UA implements some object-oriented concepts, it differs from class diagrams in some key aspects. We will therefore elaborate further on the relevant *NodeClasses* and their instantiation.

Types and InstanceDeclarations

In principle only *ObjectTypes* and *VariableTypes* can be instantiated. *Nodes* of these *NodeClasses* are also called *TypeDefinitionNodes* and the *Objects* or *Variables* that reference them as their type definition are *Instances* of the respective *TypeDefinitionNodes*. We often use *TypeDefinitionNode* and type interchangeably, but the term *TypeDefinitionNode* is a more explicit way of referring to the *Node* representing the type. *TypeDefinitionNodes* can define references to other *Nodes* which must then, depending on the *NodeClass* of the *TargetNode*, the presence of a *ModellingRule* and the *ReferenceType* of the *Reference*, also be present on their *Instances*.

In order to properly understand the instantiation of types we must first introduce the concepts of *InstanceDeclarations* and *InstanceDeclarationHierarchies*. We will sometimes restate the definitions found in the specification [OPC21b] slightly altered to better accommodate our goals or achieve a more concise definition.

Definition 2.1.1 (*InstanceDeclaration*)
An Object or Variable is called an InstanceDeclaration iff it references a ModellingRule with a HasModellingRule Reference and is the TargetNode of at least one forward hierarchical Reference where the SourceNode is either a TypeDefinitionNode or an InstanceDeclaration whose ModellingRule is Mandatory, Optional or ExposesItsArray. The type of an InstanceDeclaration may be abstract, but any instance based on it must be of a concrete type.

Note that forward hierarchical *References* are *References* of a subtype of *HierarchicalReferences*. All concrete *ReferenceTypes* are subtypes of either *HierarchicalReferences* or *NonHierarchicalReferences*.

We purposely exclude *Methods* from the definition, as we do not plan to include them in the semantic difference.
Furthermore, we distinguish between *Instances* referenced by *InstanceDeclarations* with different *ModellingRules*. This is due to the fact that *Instances* referenced by a *InstanceDeclaration* with either a *MandatoryPlaceholder* or an *OptionalPlaceholder ModellingRule* are not considered to be part of the *InstanceDeclarationHierarchy*. From here on we will sometimes refer to *InstanceDeclarations* with these *ModellingRules* as placeholders. We consider it good practice to include this in the definition of *InstanceDeclarations* because an *Instance* that would be considered an *InstanceDeclaration* solely by virtue of such a *Reference* does not differ semantically from a *Instance* that is not an *InstanceDeclaration*.
Although OPC UA allows defining new *ModellingRules*, we assume there to be only the five *ModellingRules* from the specification [OPC21b]: *Mandatory*, *MandatoryPlaceholder*, *Optional*, *OptionalPlaceholder* and *ExposesItsArray*.
Since there is currently no standard way to define the semantics of new *ModellingRules*, there is no way of taking user-defined *ModellingRules* into account without first implementing such a mechanism.

Every path from a *TypeDefinitionNode* consisting of one or more *InstanceDeclarations* connected by forward hierarchical *References* corresponds to a sequence of *BrowseNames*. This sequence is called a *BrowsePath*.

Definition 2.1.2 (*BrowsePath*)
Starting from a TypeDefinitionNode, a sequence of BrowseNames, corresponding to a sequence $(n_0, n_1, ..., n_m)$ *of Nodes, constitutes a BrowsePath. All Nodes in the path (except the start Node* n_0*) must be InstanceDeclarations.*
For $0 < i < m$ *all* n_i *must reference one of the ModellingRules Mandatory, Optional or ExposesItsArray with a HasModellingRule Reference.*
For each $i < m$ *of Nodes there must exist a forward hierarchical reference with SourceNode* n_i *and TargetNode* n_{i+1}*.*

We do not include *References* with a *MandatoryPlaceholder* or *OptionalPlaceholder InstanceDeclaration* as *SourceNode*, since these references are not considered when instantiating the type. In the context of our operator it would therefore be pointless to consider any resulting *BrowsePaths*.

Based on the definitions of *InstanceDeclarations* and *BrowsePaths* we can now introduce the aforementioned concept of an *InstanceDeclarationHierarchy*.

Definition 2.1.3 (*InstanceDeclarationHierarchy*)
The InstanceDeclarationHierarchy of a TypeDefinitionNode consists of all InstanceDeclarations that have a BrowsePath in the context of the TypeDefinitionNode. We consider forward hierarchical References between the InstanceDeclarations and from the TypeDefinitionNode to the InstanceDeclarations to be part of the InstanceDeclarationHierarchy as well.

The specification [OPC21b] is somewhat ambiguous regarding *InstanceDeclarationHierarchies* as it states that they consist only of the *InstanceDeclarations*, but at a later point declares a supporting table of *References* necessary to fully represent an *InstanceDeclarationHierarchy*. We chose to follow the latter since knowledge of the number and type of *Reference* is required for instantiating the type.

Every type inherits the *InstanceDeclarationHierarchies* of its supertypes, i.e., of all *TypeDefinitionNodes* either directly or indirectly referenced via inverse *HasSubType References*. For non-conflicting *BrowsePaths*, the *InstanceDeclarationHierarchies* are simply combined while for conflicting *BrowsePaths* the subtypes can successively override the *InstanceDeclarations*.

An *InstanceDeclaration* specified by some type can be overridden in a subtype's *InstanceDeclarationHierarchy* by an *InstanceDeclaration* with the same *BrowsePath*.

For overriding *InstanceDeclarations* some rules apply. Any *InstanceDeclaration* A overriding an *InstanceDeclaration* B

- must be of the same type as B or a subtype of B's type.
- must have the same *ModellingRule* as B or a more restrictive one. For the standard-*ModellingRules* only going from *Optional* to *Mandatory* and from *OptionalPlaceholder* to *MandatoryPlaceholder* is allowed.

A forward hierachical *Reference* r overrides another forward hierachical *Reference* r' from a supertype's *InstanceDeclarationHierarchy* if it goes between two overridden *Nodes* and r is of the same type as r' or a subtype of it. The *SourceNode* in this context can also be the *TypeDefinitionNode*.

An overriding *InstanceDeclaration* may also declare new *References*. *Reference* that are not explicitly specified are still inherited from the supertypes, but not overridden.

The *InstanceDeclarationHierarchy* together with any inherited *InstanceDeclarations* forms the so-called fully-inherited *InstanceDeclarationHierarchy*.

Instantiating a *TypeDefinitionNode* in its minimal form is very similar to instantiating classes from a class diagram. The specification [OPC21b] states that the *Instance* must be the root of a hierarchy mirroring the *InstanceDeclarationHierarchy* of the *TypeDefinitionNode*. This means that there must be *References* and similar *Nodes* present in a multiplicity determined by the *ModellingRules* of the respective *InstanceDeclaration*. A similar *Node* to an *InstanceDeclaration* is a *Node* with the same type or a subtype and the same *BrowseName*.

We will briefly explain the five standard *ModellingRules* and the corresponding rules. For each *InstanceDeclaration A* that references a *ModellingRule* and is referenced by a *TypeDefinitionNode* or a non-placeholder *InstanceDeclaration B* through one or more forward hierarchical *References*, there is a specific number of similar *Nodes* to *A* that must be referenced by any *Instance* based on *B*.

Mandatory exactly one similar *Node* must be referenced through *References* of the same *ReferenceTypes* or subtypes of these.

Optional at most one similar *Node* may be referenced through *References* of the same *ReferenceTypes* or subtypes of these.

MandatoryPlaceholder at least one *Instance* of the same type or a subtype must be referenced through *References* of the same *ReferenceTypes* or subtypes of these. These *Instances* are not required to have the same *BrowseName* as the *InstanceDeclaration*.

OptionalPlaceholder arbitrarily many *Instances* of the same type or a subtype may be referenced through *References* of the same *ReferenceTypes* or subtypes of these. These *Instances* are not required to have the same *BrowseName* as the *InstanceDeclaration*.

ExposesItsArray arbitrarily many *Instances* of the same type or a subtype must be referenced through *References* of the same *ReferenceTypes* or subtypes of these. These *Instances* are not required to have the same *BrowseName* as the *InstanceDeclaration*. In contrast to *MandatoryPlaceholder* and *OptionalPlaceholder*, for the *ExposesItsArray ModellingRule* the forward hierarchical *References* of the *InstanceDeclaration* are included in the *InstanceDeclarationHierarchy*. Unlike for *Mandatory* and *Optional*, however, the *Instances* based on such an *InstanceDeclaration* must, in the context of one specific type, all reference the same *Instance*.

This applies for *References* of *Instances* independently instantiated from a *TypeDefinitionNode* as well as for those serving as similar *Nodes* to an *InstanceDeclaration* as non-root part of a hierarchy.

Moreover, it is important to note that in OPC UA for multiple *References* connecting the same *Nodes* as part of an *InstanceDeclarationHierarchy* the *References* mirroring them in the hierarchy of any *Instance* must also connect the same *Nodes*. Otherwise, the *InstanceDeclarationHierarchy* of the *TypeDefinitionNode* would contain a *BrowsePath* that could not be mapped to a unique *Node* in the *Instances* of the *TypeDefinitionNode*.

So far we have discussed how instantiate *TypeDefinitionNodes* based on their *InstanceDeclarationHierarchies*. In OPC UA, however, *Instances* are not limited to the *References* defined by their *TypeDefinitionNode* or even those of the *InstanceDeclaration* they are based on. Except for some minor limitations, an *Instance* may have arbitrary *References* completely unrelated to its *TypeDefinitionNode*. A simple example is shown in 2.2, where an *ObjectType House* with an optional component of type *Garden* is defined. Based on this definition of the *ObjectType House* we instantiate the *Instance HouseInstance* with a component of type *JetEngine*.

As a result, when using OPC UA information models without any restrictions, the *ModellingRules* only give a lower limit of how many similar *Nodes* are expected, due to the possibility to add arbitrary *References* to a *Node*. Therefore, in standard OPC UA every *Reference* to an *InstanceDeclaration* with a *ModellingRule Optional* or *OptionalPlaceholder* can be seen as a mere suggestion, since they do not impact the possible *Instances* of the respective *TypeDefinitionNode* at all.

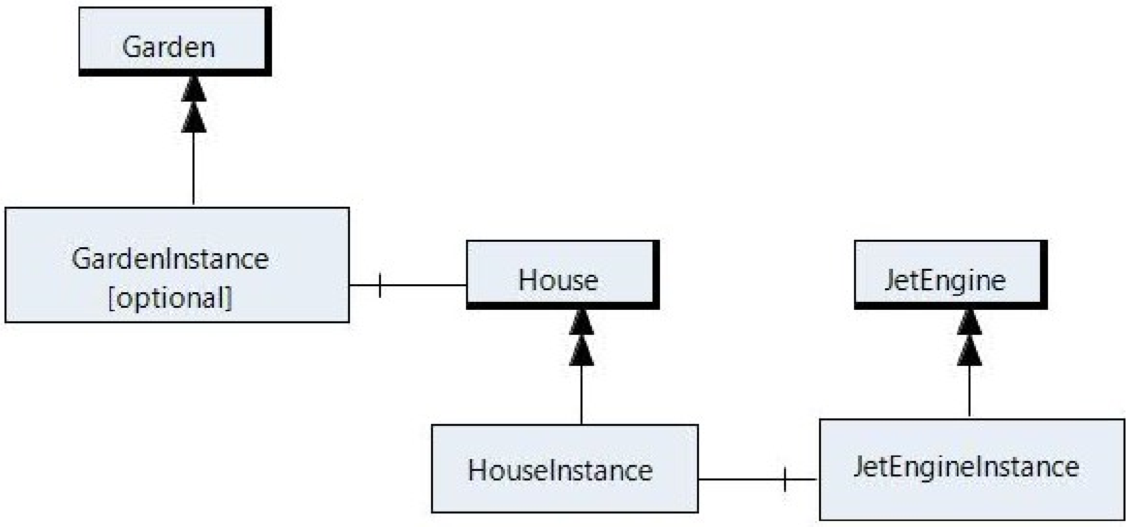


Figure 2.2: *Instance* of the House-*ObjectType* with a JetEngine component

While this arguably provides some flexibility we will propose and informally describe an alternative with stricter adherence to type definition in 3.

Interfaces and AddIns

The *InstanceDeclarationHierarchy* of a *TypeDefinitionNode* can be further expanded by referencing *Interfaces* with a *HasInterface Reference*. *Interfaces* are just *ObjectTypes* that are abstract subtypes of *BaseInterfaceType* and as such behave mostly like a normal abstract *ObjectType*.
When referenced by an *TypeDefinitionNode* with a *HasInterface Reference* all *Instances* of the type and its subtypes shall be instantiated as if all *Nodes* and *References* of the *Interface*'s *InstanceDeclarationHierarchy* were also part of the *TypeDefinitionNode*'s *InstanceDeclarationHierarchy*. *TypeDefinitionNodes* may not apply *Interfaces* with conflicting *BrowsePaths* and therefore cannot override any *InstanceDeclarations* of the *Interface*.

Additionally, OPC UA allows *Instances* to reference *Interfaces* with *HasInterface References*. In contrast to *TypeDefinitionNodes* applying an *Interface*, the *Instance* must then directly mirror the *Interface*'s *InstanceDeclarationHierarchy* as if the *Interface* were applied to the *Instance*'s *TypeDefinitionNode*.
While unusual, this feature is irrelevant for our operator as we do not concern ourselves with *Instances* that are no *InstanceDeclarations*. For *InstanceDeclarations*, however, the mirroring *InstanceDeclarationHierarchy* is also a part of any *InstanceDeclarationHierarchies* that contain the *InstanceDeclaration*.

Since it is explicitly mentioned in the specification [OPC21b] as a special *AddressSpace* concept we briefly explain our reasoning on *AddIns* for the sake of clarity.
Any *Object* can be used as an *AddIn* by referencing it from an *ObjectType* or *Object* with a forward *HasAddIn Reference*, which is as subtype of the *HasComponent* reference.

The *HasAddIn ReferenceType* has no special semantics distinguishing it from the general *HasComponent ReferenceType* other than that the *TargetNode* is called an *AddIn*. Therefore we will not handle *AddIns* in any special way and will not explicitly discuss them in later chapters.

2.1.4 MontiCore

MontiCore (MC) is described as as a "*framework for compositional development of domain specific languages*" [KRV10]. Domain specific languages (DSLs) are tailored to a specific domain, e.g., the representation of class diagrams as in CD4A [HKR21], as opposed to general purpose programming languages like Java [AGH05] or C [RY07].
In MC languages are described as context-free grammars, from which Java code is generated, along with context-conditions that can be added to the code manually.
Most importantly, the generated code includes abstract syntax tree (AST) classes, which can also be generated to be compatible with the Eclipse Modeling Framework[HKR21; SBMP11]. In combination with other generated code infrastructure such as builders and visitors for the AST classes as well as parsers for the DSL code, developers are enabled to build tools for their languages with relative ease. MontiCore as well as the associated languages and tools are available at [1].

CD4A

The language Class Diagrams for Analysis (CD4A) is a MontiCore language based on UML/P [Rum11] CDs, representing a subset of their functionality [HKR21]. UML/P is a UML profile intended for the generation of Java code. In the course of this work mentions of class diagrams will therefore refer to CD4A class diagrams unless stated otherwise.

Atop the generated code some tools have been implemented for CD4A and the near identical language Class Diagrams for Code (CD4C). We are specifically interested in the *cddiff* tool, which implements the cddiff_k operator [MRR11b]. The implementation accepts two CD4A class diagrams and a parameter k, limiting the maximum number of objects in a witness. Using MC to generate EMF-compatible AST classes allows us to leverage the powerful set of modeling-related tools provided by the Eclipse Modeling Framework, including model-to-model transformations.

2.1.5 Differencing

Differencing tools are essential for most software engineering processes as they provide precise insights on the changes between versions. Most tools approach this on a purely syntactical level, where the difference often consists of a set of add or delete operations. One of the most prominent examples for this approach is the git diff command [CS14].
In the example C++ code snippets 2.3 and 2.4, such a diff operator would detect a change in line 2 due to the changed order of a and b. While such a syntactic diff is useful, there are certain cases where we are more interested in the question whether the program produces different results, which is obviously not the case here.

[1] https://github.com/MontiCore

In order to detect changes only if they impact the semantics of the program we would need a semantic differencing tool. In the course of this work we describe and implement such a semantic differencing operator for OPC UA models.
While, to our knowledge, there is no existing semantic differencing solution for OPC UA, we can build on existing work for class diagrams.

```
int add(int a, int b) {
  return a + b;
}
```

Listing 2.3: First *add* function

```
int add(int a, int b) {
  return b + a;
}
```

Listing 2.4: Second *add* function

Syntactic Differencing

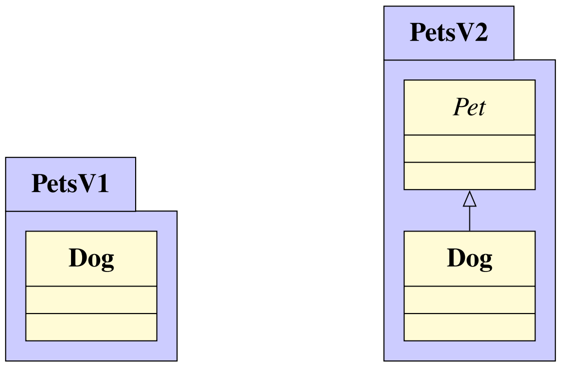


Figure 2.3: Example of two syntactically different, but semantically identical models. Adapted from [MRR11b].

Syntactic differencing approaches for models often focus on the creation, deletion and change of model elements, e.g., classes in class diagrams, between two versions [AP03; OWK03]. This offers a complete view on the changes to a model on the syntactic level. Some semantic insights, however, may not be obvious by looking at the syntactic difference.
The two class diagrams in 2.3 differ syntactically by the addition of an abstract superclass *Pet*. For both models, however, the set of possible instances is identical, consisting only of objects of the class *Dog* without any associations.
They can thus be considered semantically equivalent according to the definition from [MRR11b], which we introduce in this section. On the other hand, small syntactic changes, e.g., altering relationships in class diagrams, may lead to vastly different sets of valid instances.
To better understand these implications semantic differencing can be used.

Semantic Differencing

Intuitively the semantic differencing operator proposed by Maoz et al. [MRR10] takes two models as input and outputs a set of diff witnesses that are instances of the first but not of the second model. As it enumerates such witnesses instead of producing a description for the difference, it can be

classified as an enumerative semantic differencing operator [FALW14].
They formalize the notion by considering a modeling language $ML = \langle Syn, Sem, \text{sem}\rangle$ (as described in [HR04]), where Syn is the set of all syntactically correct expressions, Sem is a semantic domain and $\text{sem} : Syn \rightarrow \mathcal{P}(Sem)$ is a mapping from the syntactically correct expressions to subsets of the semantic domain. For two syntactically correct expressions e_1, e_2 they now define the operator $\text{diff} : Syn \times Syn \rightarrow \mathcal{P}(Sem)$ as follows:

Definition 2.1.4
$\text{diff}(e_1, e_2) = \{s | s \in \text{sem}(e_1) \wedge s \notin \text{sem}(e_2)\}$

Therefore, an element of the semantic domain is included in the difference if and only if it is an instance of the first, but not of the second input model. This straightforward but abstract definition provides a framework to build operators for specific modeling languages.
Applying this asymmetric operator in both directions can provide insights into the relationship of the two models. If both differences are empty, the models are semantically equivalent. Meanwhile, a single empty difference $\text{diff}(e_1, e_2) = \emptyset$ and $\text{diff}(e_2, e_1) \neq \emptyset$ implies that e_1 is a refinement of e_2 and e_2 is a generalization of e_1.

CDDiff

The proposed semantic difference operator for UML/P [Rum11] class diagrams, *cddiff* [MRR11b], was implemented via a reduction to Alloy.

In the class diagram context the modeling language, as introduced for the abstract operator, is $ML = \langle CD, OM, \text{sem}\rangle$. The set of syntactically correct expressions CD is the set of syntactically correct CD4A class diagrams. The semantic domain OM consists of all finite object models that can be instantiated from expressions in CD and $\text{sem} : CD \rightarrow \mathcal{P}(OM)$ maps a class diagram to the set of object models that can be instantiated from it.
They then define the operator *cddiff* as follows:

Definition 2.1.5
$\text{cddiff}(cd_1, cd_2) = \{om \in OM | om \in \text{sem}(cd_1) \wedge om \notin \text{sem}(cd_2)\}$

For practicability, since *cddiff* can be very large or even infinite, their implemented operator is a slightly modified version $\textit{cddiff}_k$, which restricts the diff witnesses to at most k instances per model. This restriction is required for the use of Alloy since the language is undecidable and solutions can therefore only be provided in a limited scope [JSS00].
The underlying assumption is an adapted version of the *small scale hypothesis*, which originally states that most flaws already occur in small instances of a model [Jac12]. The version adapted to the problem of semantic differences in class diagrams states that instances for all classes of diff witnesses can be found within a relatively small scope [MRR11b].

2.2 Related work

Regarding related work we searched mainly for semantic differencing approaches and transformations from OPC UA to UML.

2.2.1 Semantic Differencing Approaches

Non-enumerative approaches to semantic differencing produce concise definitions of the respective difference instead of a finite set of witnesses. Such operators have been described, e.g. for class diagrams [FALW14] and automata [FLW11]. As pointed out by Fahrenberg et al. [FALW14], this approach produces a complete description of the difference in the form of a model. They argue since the resulting difference resides in the same domain and retains the same level of abstraction as the input models it can be manipulated more easily by tools and engineers.
When applying this to OPC UA, it must be taken into account that the designers of new specifications are not usually software engineers but rather experts from other engineering domains. Therefore we believe an enumerative approach that provides the user with concrete witnesses to be easier to understand and use.

Enumerative approaches exist for other modeling languages such as class diagrams [MRR11b] and activity diagrams [MRR11a]. To our knowledge no semantic differencing operator of either approach has been described for OPC UA.

2.2.2 Existing Transformations

Model transformations between UML CDs and OPC UA have been described in literature [LKYO17; PWFW18; RPL13]. Combined with semantic differencing for class diagrams like *cddiff* [MRR11b] this would provide a potential easy solution, were it not for some limitations.
Existing transformations are often unidirectional from UML to OPC UA [PWFW18; RPL13] as their main goal is to facilitate the creation of OPC UA information models from UML class diagrams. A bidirectional approach has been proposed by Lee et al. [LKYO17] who professed some forced misalignment in their mapping due to OPC UA lacking a clear separation of model and meta-model elements.

None of the existing transformations we found addressed the *InstanceDeclaration* concept, including the overriding of *InstanceDeclarations*. Unidirectional approaches from UML to OPC UA need not address the issue of overriding *InstanceDeclarations* [PWFW18; RPL13]. No equivalent concept exists in UML, therefore CDs can simply be mapped to OPC UA models where no overriding of *InstanceDeclarations* occurs.
Even the bidirectional approach [LKYO17] does not include the concept, as *InstanceDeclarations* are mapped to the same classes as their *TypeDefinitionNode*.
While the CDs resulting from such a transformation are better suited for further use than the ones we create, they do not reflect OPC UA semantics as accurately as required for meaningful semantic differencing.
Furthermore, our target language is CD4A, which would require either a second transformation or rewriting the existing one for CD4A, if we were to use one.

2.2.3 Literature Research Methodology

For our literature research the starting point were the papers about *cddiff*, the underlying enumerative semantic differencing approach and its implementations [MRR11a; MRR11b; MRR12; MRR14].

From there we conducted the research mainly via the *Google Scholar* [2] search engine. Used keywords include *differencing, diffference, semantic, syntactic, transformation, mapping, UML, cd, class diagram, cd4a, UA, OPC UA* and combinations of those. Additionally, we followed relevant citations of sources to discover related work. For some topics we also searched in the university library catalogue of the University of Stuttgart [3].

We tried to use papers that were published in some peer-reviewed form when possible, but especially regarding semantic differencing and general transformations between OPC UA and UML, the number of related papers is rather small.

To assess the usefulness of a paper for our work, we first read the abstract and based on that decided whether to read the rest of the paper.

When referenced in some paper or concerning an important tool, we also included some books, e.g., for UML/P [Rum11], MC [HKR21] or EMF [SBMP11].

[2] https://scholar.google.com/

[3] https://www.ub.uni-stuttgart.de/en/

3 UADiff Operator

We define our operator based on *cddiff* [MRR11b] by mapping OPC UA models to CD4A class diagrams and then applying the existing operator.
Reusing *cddiff* offers some advantages, a more obvious one being the reuse of a quite complex piece of software instead of implementing a very similar one from scratch. Additionally we can build on the well-defined semantics of CD4A [CGR14], defining a transformation UA2CD in ATL, thus obtaining translational semantics [CK07] for OPC UA models.

As a modeling language in the sense of [HR04] (see section 2.1.5), we can define OPC UA as follows:

Definition 3.0.1 (ML_{UA})
$ML_{UA} := \langle UA, OM, \mathrm{sem}_{\mathrm{UA}} \rangle$

The syntactic domain UA consists of all syntactically correct XML *NodeSets* conforming to OPC UA XSD [OPC21c] and the constraints stated in [OPC21b].
We define the semantic mapping as the composition of the UA2CD : $UA \rightarrow CD$ mapping and a slightly modified version of the semantic mapping for CDs used in [MRR11b], sem' : $CD \rightarrow \mathcal{P}(OM)$. We explain our modifications to the CD semantic mapping and our reasons for applying them in section 3.2.

Definition 3.0.2 ($\mathrm{sem}_{\mathrm{UA}}$)
$\mathrm{sem}_{\mathrm{UA}} : UA \rightarrow \mathcal{P}(OM), n \mapsto \mathrm{sem'} \circ \mathrm{UA2CD}(n)$

The operator *uadiff* is then defined as follows:

Definition 3.0.3 (uadiff)
$\mathrm{uadiff}(n_1, n_2) := \{om \in OM | om \in \mathrm{sem}_{\mathrm{UA}}(n_1) \land om \notin \mathrm{sem}_{\mathrm{UA}}(n_2)\}$

Note that the actual implementation described in chapter 4 is an operator *uadiff*$_k$ with a maximum of k objects in any witness, analogous *cddiff*$_k$.

3.1 Mapping UA to CD

With our mapping UA2CD we map a set of OPC UA *Nodes* and *References* to a CD4A class diagram. In addition to simply enabling the use of *cddiff*, there are several restrictions we impose on the instantiation of OPC UA types, that are reflected in this mapping.
We give an overview of the mapping in table 3.1. In this chapter we describe CD4A in terms of the represented concepts, whereas in the implementation of the mapping in chapter 4 CD4A diagrams are represented through their ASTs.

OPC UA Element	**CD4A Element**
TypeDefinitionNode	*Interface* and *Class*
InstanceDeclaration	*Classes*
HasSubtype Reference	*Inheritance* relationship
other *forward hierarchical Reference*	*Association*
other *Nodes* and *References*	*none*, but may provide context information

Table 3.1: Mapping from OPC UA to UML

3.1.1 Assumptions about CDDiff

We use an older version of CD4A where some context conditions are not implemented. This leads to CDs being valid although a subclass defines an association with the same name as an association that is already present in a superclass. Due to the way *cddiff* is implemented this enables us to refine associations between subclasses regarding multiplicity and which subclass the specific side belongs to.

The described behaviour can be explained using the CD in figure 3.1:
Every *Animal* eats an arbitrary amount of *Fruits*, while each *Fruit* is eaten by at most one *Animal*. *Apes*, however, only eat *Bananas* and at most one. *Bananas* are still eaten by at most one *Animal*, but only if it is an *Ape*.

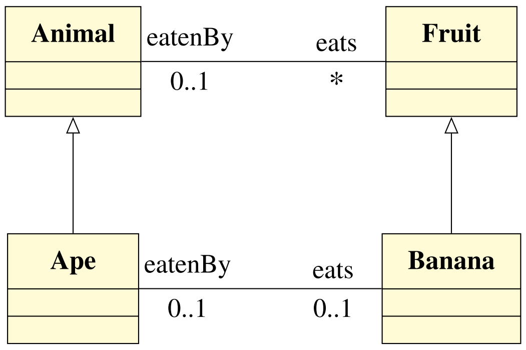


Figure 3.1: Subclasses with a refined association.

3.1.2 Nodes

We map three different classes of *Nodes* to some CD construct: *Interfaces*, non-interface *TypeDefinitionNodes* and *InstanceDeclarations*.

For an *Interface* we simply create a CD interface. *Interfaces* cannot be instantiated and instances of any type that implements them must mirror the *InstanceDeclarationHierarchies* of the respective *Interfaces*. Furthermore, *ObjectTypes* can implement multiple interfaces. The only CD construct with the described characteristics is an interface, to which we therefore map our *Interfaces*.

Any *TypeDefinitionNode* that is not an *Interface* is mapped to an interface and a class that implements the interface. Classes are abstract iff the respective *TypeDefinitionNode* is abstract. The type's *References* are mapped to associations of the interface, not the class. This allows us to handle the *InstanceDeclarations* more easily.

While *TypeDefinitionNodes* are quite straightforward to map to CD constructs, *InstanceDeclarations* are rather complicated.
InstanceDeclaration with *ModellingRules Mandatory*, *Optional* or *ExposesItsArray*, can specify new *References* or override those given by the *TypeDefinitionNode*. Additionally, for *InstanceDeclarations* with *Mandatory* or *Optional ModellingRules* the *Instances* must have the same *BrowseName*. Therefore we need to map each of these *InstanceDeclarations* to a class.
Any *InstanceDeclaration* with a *ModellingRule MandatoryPlaceholder* or *OptionalPlaceholder*, on the other hand, correspond exactly to their *TypeDefinitionNode*. For those *InstanceDeclarations* neither a *BrowseName* nor any *References* beyond those specified by their *TypeDefinitionNode* are mandated. We map them to classes the same way as other *InstanceDeclarations* because the overriding mechanism described in the previous section 3.1.1 requires subclasses of the original classes on both sides of the association.
For convenience we assume an *InstanceDeclaration* ID and a *TypeDefinitionNode* T, where ID is of type T. The class ID_{class} based on ID is a subclass of the class T_{class}, to which T is mapped. We also want to reflect that *Instances* based on *InstanceDeclarations* may have the same type as the *InstanceDeclaration* or a subtype of it. For this purpose we create additional classes, mirroring the subtype hierarchy of T. We give an example using the OPC UA model in figure 3.2. The class hierarchy created from the *ZooAnimal InstanceDeclaration*, including associations, extended classes and implemented interfaces, is shown in figure 3.3.

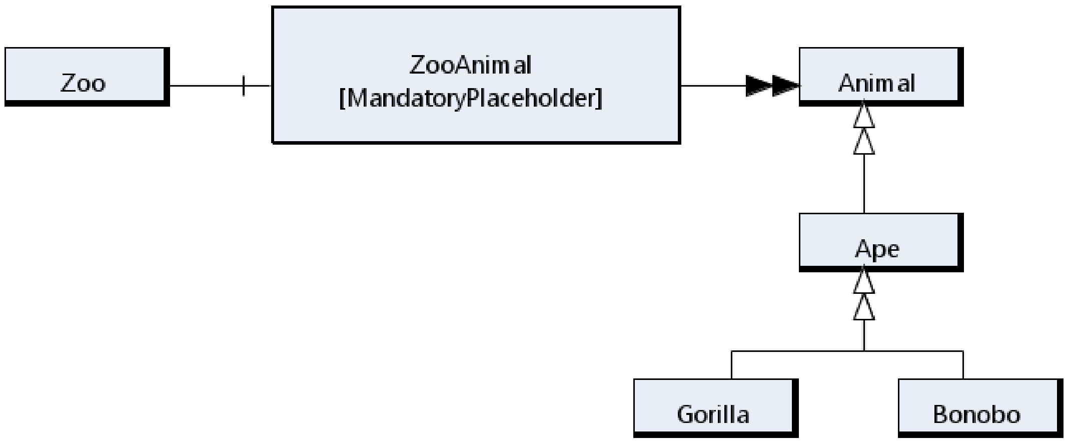


Figure 3.2: OPC UA model including an *InstanceDeclaration* of type *Animal*, which has several subtypes.

This has the severe drawback of quickly leading to immensely large CDs, due to the large subtype hierarchies in OPC UA. We address this issue in chapter 4 by considering only a user-specified subset of the given *NodeSet*. The structure of the inheritance relationships provides another challenge,

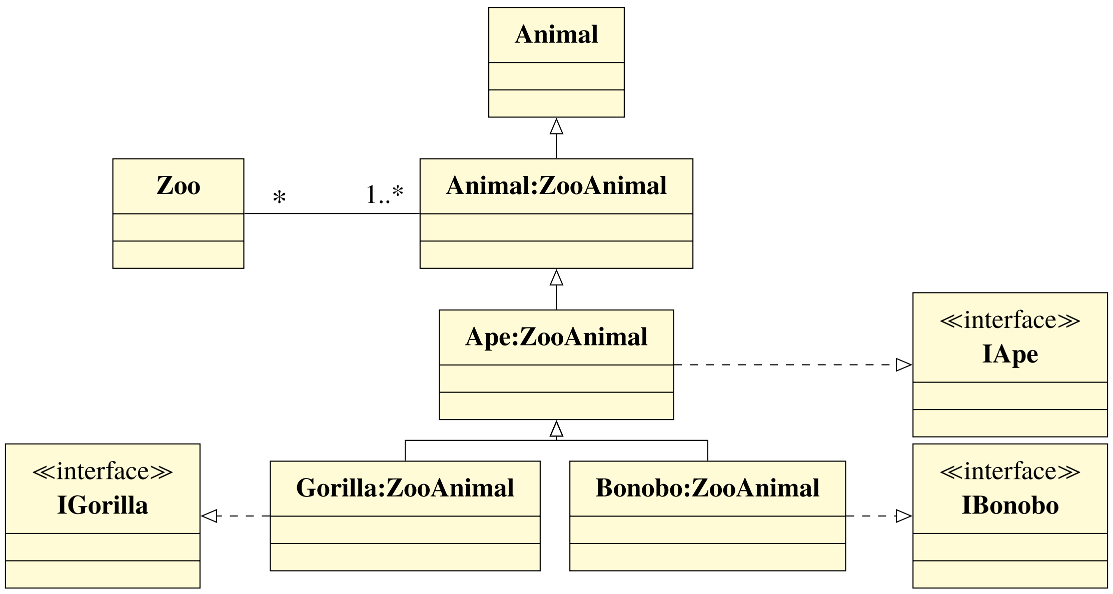


Figure 3.3: Class hierarchy created for the *InstanceDeclaration ZooAnimal* from figure 3.2.

since the root of the created subclass-hierarchy, ID_{class}, is already a direct subclass of T_{class} and can therefore not inherit from any other direct subclass of T_{class}. For this purpose we map the *TypeDefinitionNodes* not only to classes, but also to interfaces. Each class in the created subclass hierarchy can therefore inherit the interface corresponding to the *TypeDefinitionNode* it is based on. A special case is the *InstanceDeclaration* ID overriding another *InstanceDeclaration* ID' of type T'. In such a case the mapping is identical except for the superclass of ID_{class}, which would then be either ID'_{class} if $T' = T$ or otherwise the class in the hierarchy of ID'_{class} corresponding to T. Such a class must be present, since ID can only override ID' if T is T' or a subtype of T'.

In most cases this mapping is sufficient, but in OPC UA an *InstanceDeclaration* could in theory be part of multiple *InstanceDeclarationHierarchies* and override a different *InstanceDeclaration* in each *InstanceDeclarationHierarchy*. Such a construct would necessitate a separate hierarchy of classes for each context the *InstanceDeclaration* is a part of. Due to time limitations we choose not to handle such cases in our implementation. Moreover, it can be argued that modeling types in such a way constitutes bad practice, since the resulting *NodeSets* would be potentially confusing for developers and users.

Additionally, all classes created from *InstanceDeclarations* are private, while classes created from *TypeDefinitionNodes* are not. We use this as a convenient way for the modified semantic mapping to distinguish between those classes, while for the original cddiff operator UML modifiers are irrelevant.

Nodes of other *NodeClasses* are not mapped since they are not instantiable, with the exception of *Methods*. Including *Methods* in a meaningful way would exceed the scope of this work.

3.1.3 References

References are mapped depending on their *ReferenceType*. Additionally we often need to distinguish *References* based on their direction. While this may seem like a purely technical aspect better left for the implementation chapter, the direction in which a *Reference* is given is often important. Some *References* might have a mandatory and an optional direction, e.g., for *HasSubType* only the inverse *Reference* is always present. For hierarchical *References* only the forward ones are considered when instantiating a *TypeDefinitionNode*.
Since we only map *TypeDefinitionNodes* and *InstanceDeclarations* to classes, *References* are only mapped if they go between two such *Nodes*

For nonhierarchical *ReferenceTypes* we map a forward *HasInterface* reference to a realization relationship if and only if the *SourceNode* is an *ObjectType*. The class the *SourceNode* is mapped to shall then reference the interface the *TargetNode* is mapped to.
If the *SourceNode* is an *Object* we do not need to apply the respective interface to the class since the *Object* must mirror the *Interface*'s *InstanceDeclarationHierarchy* and only the *Nodes* appearing in the mirrored hierarchy are relevant for instantiation. If the *Object* is not an *InstanceDeclaration* it is not relevant for us anyway.
Other nonhierarchical *References*, e.g., *HasModellingRule* or *HasTypeDefinition* may provide additional information for *Nodes* or other *References*, but they are not directly relevant for the instantiation of *TypeDefinitionNodes* and are therefore not mapped to any concrete CD element.

Hierarchical *References*, on the other hand, are mapped to specific relationships, depending on their *ReferenceType* and direction.
Inverse *HasSubType* references shall be mapped to inheritance relationships, as subtyping in OPC UA is quite similar to inheritance in CDs. The overriding of *References* and the concept of *InstanceDeclarations* has no counterpart in CDs, but we handle this by mapping *InstanceDeclaration* accordingly. The resulting inheritance relationship shall go between the class corresponding to the *TargetNode* as superclass and the one created from the *SourceNode* as subclass.
We ignore forward *HasSubType References* since they are completely optional, while the inverse *References* are always present.
Other subtypes of *HierarchicalReferences* are mapped to standard associations. In the implementation the exact *ReferenceType* will be reflected in the name of the association. There are some constraints associated with specific *ReferenceTypes*:

1. *Properties*, i.e., *TargetNodes* of forward *HasProperty References*, shall not be *SourceNodes* of hierachical *References*
2. *ReferenceTypes* may restrict what *NodeClasses* are permitted as *SourceNodes* or *TargetNodes*.
3. *References* of the *HasChild ReferenceType* or any of its subtypes shall never form a cycle. The same goes for *References* of the *HasEventSource ReferenceType* and its subtypes.einhalten

Provided the input models are valid, constraints 1 and 2 are trivially fulfilled in the context of our work as we do not allow arbitrary *References* not specified by the *TypeDefinitionNode* on *Instances*. A *Property* with forward hierarchical *References* would therefore have to be instantiated from an *InstanceDeclaration* that is also a *Property* with forward hierarchical *References*, i.e., from an invalid model.
For the second constraint, if an *Instance* is instantiated from an *Object* or *ObjectType* it is always an *Object* and when instantiated from a *Variable* or *VariableType* it is always a *Variable*.

Therefore, the only problematic cases would be *Instances* of *ObjectTypes* or *Variables* where the *Reference* is restricted to only the respective *NodeClass*. For hierarchical *ReferenceTypes* defined in [OPC21b] this is only the case with *HasSubType*, which is not part of *InstanceDeclarationHierarchies*. If another hierarchical *ReferenceType* would be constrained in this way we could safely consider any *InstanceDeclarationHierarchy* containing such a *Reference* invalid as there could never be a valid *Instance* mirroring the *Reference*.
The third constraint is not reflected in our operator. Due to our *complete* interpretation of OPC UA models such a case would only be possible for a valid model if the fully inherited *InstanceDeclarationHierarchies* of two *TypeDefinitionNodes* contain *InstanceDeclarations* of the respective other type. Such cases should be rare because of the typically hierarchical design of OPC UA information models.

Information regarding the cardinality resides not in the *References* but in the different *InstanceDeclarations* via a *Reference* to a *ModellingRule*. These *ModellingRules* are mapped as shown in table 3.2 according to [OPC21b] except for *ExposesItsArray*, for which no such cardinality is specified. We selected the multiplicity of zero or more as the closest match as the version of CD4A we use in our implementation does not allow upper bounds other than 1. In reality the number of *Instances* based on a *ExposesItsArray InstanceDeclaration* referenced by a single *Node* may be limited by the *ValueRank* and *ArrayDimensions Attributes* of the *InstanceDeclarations*, while the *ModellingRule* in itself does not imply any cardinality restricitons. Also note the property of *ExposesItsArray* that similar *Nodes* of *InstanceDeclarations* must reference the same *Nodes* with all their forward hierarchical *References* if they belong to the same *InstanceDeclarationHierarchy*. This property is not considered by *uadiff*.

OPC UA *ModellingRule*	**CD4A Multiplicity**
Mandatory	1
Optional	0..1
MandatoryPlaceholder	1..*
OptionalPlaceholder	*
ExposesItsArray	*

Table 3.2: *ModellingRule* of the *Reference*'s *TargetNode* in OPC UA mapped to UML multiplicities. The mapping for all *ModellingRules* except *ExposesItsArray* is found in [OPC21b].

The specified cardinalities are always applied on the association side of the class corresponding to the *TargetNode* of the mapped forward hierarchical *Reference*. Since the *TargetNodes* are *InstanceDeclarations* this refers to the root of the class hierarchy we map the specific *InstanceDeclaration* to. The other association side always has cardinality $*$, as in OPC UA there is no restriction on how many *Nodes* can be *SourceNodes* of forward hierarchical *References* to a single *Node* except for *HasSubType*, which does not concern us here as it is only mapped to inheritance relationships. This is true even for *References* of the *HasComponent ReferenceType*, which is why a mapping to composition is not warranted. While aggregation may better reflect the intended meaning of *HasComponent References* to a human reader, we decide on the use of standard associations for simplicity as regards the implementation.

We consider associations to go from left to right regarding the direction of the corresponding *Reference*, i.e., the left side of the association is always the class created from *SourceNode* and the right side is always the class corresponding to the *TargetNode*. We then use this property in our modified semantic mapping to determine the direction.

If an *InstanceDeclaration* is overridden, the overriding *InstanceDeclaration* may reference a different *ModellingRule* that is applied on instantiation. *References* are only overriden if the *TargetNode* is overridden and the *SourceNode* is either the *TypeDefinitionNode* or also overridden. Therefore, the association created from the overriding *Reference* always connects two subclasses of the ones connected by the association created by the overridden reference. Moreover, OPC UA allows only tightening of constraints when overriding *InstanceDeclarations*. For an overriding reference, the multiplicity of the created association is thus refined accordingly as shown in figure 3.1.
The allowed *ModellingRules* for overriding an *InstanceDeclaration* with a specific *ModellingRule* are shown in 3.3. Input models are considered valid only if they satisfy these constraints.

ModellingRule* of the overridden *InstanceDeclaration	***ModellingRule* of the overriding *InstanceDeclaration***
Mandatory	*Mandatory*
Optional	*Optional* or *Mandatory*
MandatoryPlaceholder	*MandatoryPlaceholder*
OptionalPlaceholder	*OptionalPlaceholder* or *MandatoryPlaceholder*
ExposesItsArray	*ExposesItsArray*

Table 3.3: Rules for changing *ModellingRules* when overriding *InstanceDeclarations* from [OPC21b]. *ExposesItsArray* does not appear in the list for allowed changes and has different semantics to the other *ModellingRules*. We therefore assume it to always remain unchanged.

An ambiguous situation as regards the overriding of an *InstanceDeclaration* ID of type T can occur when a subtype T_{sub} of T and ID both have forward hierarchical *References* to *InstanceDeclarations* with some *BrowseName* B. The *References* in this scenario are of the same type or one is a subtype of the other. When creating an *Instance* of type T_{sub} from ID we have to decide if either T_{sub} takes precedence over ID or the other way round in terms of the *Reference* the *Instance* has to mirror. This is in regard to the *ReferenceType* of the *Reference*, the *TargetNode* and the *ModellingRule* of the *TargetNode*.
In our interpretation ID can only be instantiated with type T_{sub} if the *TargetNodes* of such *References* are either identical or neither of them have a *Mandatory ModellingRule*, as the two different *TargetNodes* would be mapped to separate classes where none is a subclass of the other.

In its current version our operator does not support every aspect of overriding in OPC UA we would like to include. Specifically, in order to override a *Reference* the overriding *Reference* must be of the same *ReferenceType*. If the *ReferenceType* is a subtype we currently do not consider the *Reference* as overriding.

3.1.4 Reflected Restrictions

For cddiff a "*complete interpretation for CDs*" [MRR11b] is used, i.e., there are no elements in the object model that are not defined in the CD (see also [Rum11]). By mapping OPC UA models to CDs such that the resulting CDs contain only associations that correspond to *References* in type definitions, we apply this complete interpretation to OPC UA models.
Every class we create corresponds to either a *TypeDefinitionNode* or an *InstanceDeclaration* and every association is created from an *Reference*. Therefore any object in an output model of *uadiff* can be considered an instance of a specific *TypeDefinitionNode* or *InstanceDeclaration* present in the first input model Also any link on such an object corresponds to a *Reference* where the respective *Node* is either the *SourceNode* or the *TargetNode*.

This complete interpretation of OPC UA models is not only necessary to allow meaningful semantic differencing, but we also consider it to be well suited for the design of OPC UA models. The possibility of adding arbitrary *References* to *Instances* provides more flexibility, e.g., for users who need to extend an *Object* representing a machine by a new component, but do not wish to define their own type. From a modeling point of view, however, the expressiveness of such models is severely limited, as *References* introduce at most a lower bound for the cardinality. While this presents a way to specify minimum requirements for *Instances* of a *TypeDefinitionNode*, concerning the intended use of a type it is not binding in any way.
We refer to chapter 2 for our example of a *House* type with an optional *Garden* 2.2. Although, in theory, a designer may have intended for houses to be equipped with jet engines, we believe that for the purpose of a model design process, houses are better assumed to come with gardens.
In contrast, when using a complete interpretation, the model in figure 2.2 describes a precise set of possible *Instances*, as shown in figure 3.4, in an easy to understand way.

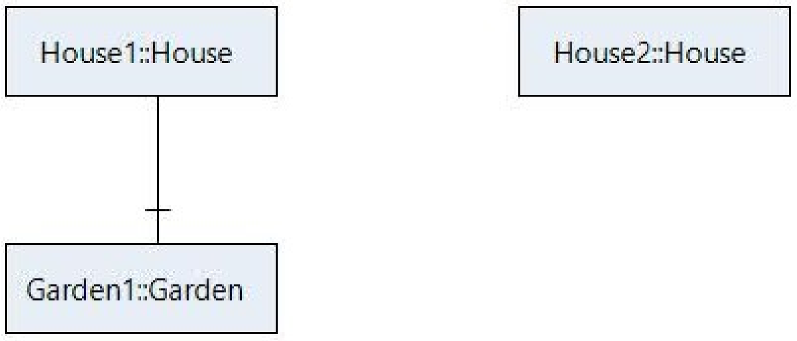


Figure 3.4: All possible variations of *Instances* for the *House ObjectType* in figure 2.2, using a complete interpretation of OPC UA models.

3.2 Changes to CD4A's semantic mapping

In order to properly reflect the instantiation of OPC UA models, a mere mapping to CDs proves insufficient. Therefore we introduce two changes regarding the semantic mapping sem used in [MRR11b], resulting in the modified semantic mapping sem'. We will not formally define sem', but give an informal account and in chapter 4 go into detail on the changes to the generated Alloy modules.

The first important change concerns public classes, i.e., the classes *InstanceDeclarations* are mapped to. We restrict the semantic mapping in such a way that an object model is only valid if all instances of public classes are directly or indirectly connected to an instance of a non-public class via links navigated from right to left. Since associations always correspond to a forward hierarchical *Reference* and the left side always corresponds to the *SourceNode*, this restriction ensures that *InstanceDeclarations* are only instantiated as part of a *TypeDefinitionNode*'s *InstanceDeclarationHierarchy*.
InstanceDeclarations may contain *References* not specified by their *TypeDefinitionNode* but they are not actual subtypes. In order to maintain the complete interpretation of OPC UA models, we do not allow such pseudo-subtypes to be instantiated outside their specific context, namely the *InstanceDeclarationHierarchies* they are part of. This way only the actual *TypeDefinitionNodes* are instantiated without being part of some other *TypeDefinitionNode*'s *InstanceDeclarationHierarchy*. At the same time objects based on *InstanceDeclarations* are still possible if they are part of a respective mirrored *InstanceDeclarationHierarchy*.

The second change concerns mutliple associations between the same classes, including inherited associations. If two classes are connected by more than one association in the same direction, every pair of objects instantiated from these classes must either be connected by links corresponding to all these association or not connected by any such link. Through this rule we enforce the analogous OPC UA constraint that *Instances* of *InstanceDeclarations* directly connected by multiple *References* must also be connected by all those *References*.

4 Implementation

In this chapter we discuss the actual implementation of the described operator in detail. Along with the more technical and precise description of the transformations we review the application architecture and state of implementation.

4.1 ATL Transformation

In chapter 3 we discussed how our operator builds on *cddiff*. We gave an informal account of OPC UA model semantics in the context of this work and how this is realized with the transformation from OPC UA to CD4A. In this chapter we go into detail regarding the actual ATL rules and some helpers.

4.1.1 Helpers

The ATL module implementing the transformation uses a variety of helpers. Most helpers are used to navigate structures within the OPC UA *NodeSet* and are usually fairly self-explanatory. Some helpers, however, relate to aspects of this specific transformation, rather than OPC UA models in general. Therefore, we go into detail for the high-level helpers regarding naming of classes and determining the relevant subset of the source elements for the transformation.

Relevant Subset

The implementation of *cddiff* cannot handle large CDs due to the generated Alloy module and thus the Boolean formula quickly growing in size. Since SAT is an NP-hard problem such large formulas can quickly increase the execution time beyond what is viable for practical application. We therefore expect two sets of *NodeIds* as input to our tool:

Included Nodes are the set of *TypeDefinitionNodes* we are interested in. If an *InstanceDeclaration* is of one of these types, we generate all the respective classes, including those corresponding to any subtypes. In many OPC UA models, there are *InstanceDeclaration* of very general types, e.g., *BaseObjectType* or *BaseVariableType*, that of course have a plethora of subtypes. Since including all those subtypes results in very large CDs, we ignore any subtypes of not included *Nodes* if they are not relevant for some other reason.

Independent Nodes are the *TypeDefinitionNodes* that are instantiated independently and not solely as part of an *InstanceDeclarationHierarchy*. All subtypes of the explicitly specified independent types are treated as such as well. The independent *Nodes* are always a subset of the included *Nodes*. By specifying independent *Node* the possible instances can be limited such that they are of the types the user is actually interested in. Thus, the purpose of this set is not increasing performance, but providing more meaningful witnesses as output.

Within the transformation, *TypeDefinitionNodes* are checked whether they are included in the typically small sets. This is accomplished by means of methods *isIncluded()* and *isIndependent()*, provided through a native Java helper.
Note that, while automatically including all subtypes of the explicitly specified types in the respective sets makes testing the operator by hand easier, when using it with some other application, e.g., an OPC UA modeling tool, one may wish to limit the sets to the types that are given explicitly.

The included *Nodes* as described above do not form a valid OPC UA model. The global attribute helper *relevantNodes* 4.1 determines all *Nodes* that must be included to form a valid model and is used in the matched rules to ensure only those *Nodes* are matched. Its value is a map from all *TypeDefinitionNodes* and *InstanceDeclarations* to a Boolean value, signifying whether the *Node* is relevant.
To construct this map, the helper *findAllRelatedNodes()* 4.2 is used with the included types and their subtypes as input. In *findAllRelatedNodes()* all supertypes and implemented interfaces of the *types* argument are collected, their union with *types* being *allTypesAndInterfaces*. For this set, we collect all *Nodes* contained in their *InstanceDeclarationHierarchies*, *hierarchyMembers*, and collect their types, *idTypes*. If *idTypes* contains types that are not included in *allTypesAndInterfaces* or considered in previous iterations, i.e., included in *seenTypes*, *findAllRelatedNodes()* calls itself recursively. The new *types* are now only the newly found types of *InstanceDeclarations* and the recursion stops when no *InstanceDeclarations* of a previously unseen type are found.
The return value is always the union of *hierarchyMembers*, *allTypesAndInterfaces* and, if present, the next recursion step's return value.

By determining the relevant *Nodes* this way, we ensure that for every type that may be instantiated the fully inherited *InstanceDeclarationHierarchy* is part of the selected subset. The constructed set is the minimal set S satisfying the following conditions:

1. S contains all *included TypeDefinitionNodes* and their subtypes.
2. If a *TypeDefinitionNode* t is contained in S, then S contains all supertypes and implemented *Interfaces* of t
3. If a *TypeDefinitionNode* t is contained in S, then S contains all *InstanceDeclarations* contained in the *InstanceDeclarationHierarchy* of t
4. If an *InstanceDeclaration* i is contained in S, then S contains the *TypeDefinitionNode* referenced by i with a *HasTypeDefinition Reference*

Note that *Interfaces* in OPC UA are also *TypeDefinitionNodes* and therefore all supertypes of *Interfaces* in S are contained in S as well.
Special care must be taken for *InstanceDeclarations* of an abstract type. Our construction of the set of relevant *Nodes* does not guarantee that a non-abstract subtype is included, even if such a type exists in the input model.

Many relevant *TypeDefinitionNodes*, while needed in order to retain complete *InstanceDeclarationHierarchies*, are not *Nodes* that should be instantiated. To give an example, *BaseObjectType* will be a relevant *Node* in most cases, as every *ObjectType* must be a subtype of *BaseObjectType*.

The *isAbstract()* helper is used to determine if a *TypeDefinitionNode* should actually be instantiated. In contrast to the *isAbstract Attribute* of *TypeDefinitionNodes*, the result of this helper is never applied to classes based on *InstanceDeclarations*. Internally, a call of *isAbstract()* looks up the value stored for the *TypeDefinitionNode* in *abstractTypesMap* 4.3.
For this map we collect all independent *TypeDefinitionNode* and all their subtypes. For every relevant *TypeDefinitionNode* the value in *abstractTypesMap* is true iff it is not included in the union of these two sets or its *isAbstract Attribute* is true.

Due to our modification of *cddiff*, abstract classes for concrete *TypeDefinitionNodes* are technically not needed, as any non-independent *TypeDefinitionNode* is mapped to a public class. If this public class is not on the right side of any association, which is the case for classes created from *TypeDefinitionNodes*, it can never be instantiated. However, we believe that additionally marking these *Classes* as abstract is a more concise way to convey the intention of excluding some *TypeDefinitionNodes*, e.g., those of very general supertypes, from being instantiated at all.

```
helper def : relevantNodes : Map(UA!UANode, Boolean) =
    let relevantNodes : Set(UA!UANode) =
        thisModule.findAllRelatedNodes(thisModule.includedTypesAndSubTypes, Set{})->union(thisModule.
includedTypesAndSubTypes)
    in let typesAndInstances : Set(UA!UANode) =
        UA!UAObjectType.allInstances()->asSet()->union(UA!UAVariableType.allInstances()->asSet())->union(thisModule.
instanceDeclarations)
    in
        typesAndInstances->iterate(node ; map : Map(UA!UANode , Boolean) = Map{} |
            map.including(node, relevantNodes.includes(node))
        )
    ;
```

Listing 4.1: ATL *relevantNodes* atribute helper.

```
helper def : findAllRelatedNodes(types : Set(UA!UANode), seenTypes : Set(UA!UANode)) : Set(UA!UANode) =
    let allTypes : Set(UA!UANode) = types
        ->collect(type | type.getAllSuperTypes())->flatten()->asSet()->union(types)
    in let interfaces : Set(UA!UANode) = allTypes
        ->collect(type | type.getInterfaces())->flatten()->asSet()
    in let allInterfaces : Set(UA!UANode) = interfaces
        ->collect( interface | interface.getAllSuperTypes())->flatten()->asSet()->union(interfaces)
    in let allTypesAndInterfaces : Set(UA!UANode) = allTypes->union(allInterfaces)->asSet()
    in let hierarchyMembers : Set(UA!UANode) = allTypesAndInterfaces
        ->collect(type | type.getInstanceDeclarationHierarchy(Map{}, Set{},'')->getValues())->flatten()->asSet()
    in let currentSeenTypes : Set(UA!UANode) = seenTypes->union(allTypesAndInterfaces)
    in let idTypes : Set(UA!UANode) = hierarchyMembers->collect(id | id.getTypeDefinitionNode())->asSet()-
currentSeenTypes
    in
        if idTypes.isEmpty()
        then
            hierarchyMembers->union(allTypesAndInterfaces)->asSet()
        else
            thisModule.findAllRelatedNodes(idTypes, currentSeenTypes)
                ->union(hierarchyMembers)
                ->union(allTypesAndInterfaces)
                ->asSet()
        endif
;
```

Listing 4.2: ATL *findAllRelatedNodes* helper.

```
helper def : abstractTypesMap : Map(UA!UAType, Boolean) =
    let relevantNodes : Set(UA!UAType) = thisModule.relevantNodes->getKeys()->select(key | thisModule.relevantNodes->
get(key))
    in let independentTypes : Set(UA!UAType) =
        UA!UAType.allInstances()->select( type | type.isIndependent())->asSet()
    in let typesAndSubtypes : Set(UA!UAType) =
        independentTypes->union(
            independentTypes->collect(type | type.getAllSubtypes())->flatten()->asSet()
        )->asSet()
    in let types : Set(UA!UAType) = relevantNodes->select(node |node.oclIsKindOf(UA!UAType))->asSet()
    in
        types->iterate( type ; map : Map(UA!UAType , Boolean) = Map{} |
            map->including(
                type,
                type.isAbstract or (not typesAndSubtypes->includes(type))
            )
        )
    ;
```

Listing 4.3: ATL *abstractTypesMap* attribute helper.

Naming

The *getClassName()* helper 4.4 is the foundation of our naming approach, using the *BrowseName* of the given *Node* as basis for the resulting class name. Other conceivable choices to use as a basis for class names are the *NodeId* and the *DisplayName*. The former is unique within an *AddressSpace* and would therefore be an excellent choice if it did not consist of a numeric *nameSpaceIndex* and an identifier, in XML *NodeSets* also a numeric value. The latter is intended for display to the user but comes with several drawbacks. It may come in different localized versions, which would require the transformation to select a specific locale. More importantly, in contrast to the *BrowseName* it is not guaranteed to be unique within the context of an *InstanceDeclarationHierarchy*, making it more difficult to handle. *BrowseNames* are unaffected by these drawbacks and are usually still suitable for display.

The helper first creates a *baseName* by removing some reserved characters from the *BrowseName* to avoid creating an invalid Alloy module. If the *isTypeToInterface* argument is true, a leading *I* is added to the *baseName* to create a name for a CD interface based on a non-*Interface TypeDefinitionNode* (see rule 4.6). If the resulting name would not be unique the *NodeId* is appended to ensure uniqueness. Different rules might modify the name further for some classes. For example, classes created from *InstanceDeclarations* have their *TypeDefinitionNode*'s class name prepended.

The *getSuperTypeName()* helper returns the name of the superclass for any class created with the *UAInstanceDeclaration2Class* rule A.1. If the *InstanceDeclaration* overrides another *InstanceDeclaration* that superclass is a class created from the overridden *InstanceDeclaration*. Otherwise, the name of the class created from the *InstanceDeclaration*'s *TypeDefinitionNode* is returned. For an *InstanceDeclaration* A overridding an *InstanceDeclaration* A', however, the returned superclass name is the name of the class created from A' that corresponds to the type T_A of A, which may be the same type as A' or a subtype. Due to our naming scheme, this is accomplished by simply applying *getClassName(*A'*)* and prepending *getClassName(*T_A*)* via *prependTypeNameToInstanceDeclaration(*T_A*)*. The branch for *InstanceDeclaration* with multiple overridden *InstanceDeclaration* additionally appends the class name of the *InstanceDeclarationHierarchy*'s root type, but is unused as the transformation is not fully implemented for such structures at the time of writing.

```
helper context UA!UANode def : getClassName(isTypeToInterface : Boolean) : String =
    let baseName : String = 'UA' + self.browseName.removeReservedChars() -- quick fix: Alloy does not allow leading numbers in names
    in
        if not isTypeToInterface
        then
            if thisModule.nonUniqueBrowseNames->includes(self.browseName)
            then
                baseName.appendNodeId(self)
            else
                baseName
            endif
        else
            if thisModule.browseNamesWithConflictingIName->includes(self) or thisModule.nonUniqueBrowseNames->includes(self)
            then
                baseName.appendNodeId(self).typeNameToInterfaceName()
            else
                baseName.typeNameToInterfaceName()
            endif
        endif
    ;
```

Listing 4.4: ATL *getClassName* helper.

```
helper context UA!UAInstance def : getSuperTypeName(hierarchyRoot : UA!UAType) : String =
    let overriddenIds : Set(TupleType(root : UA!UAType, id : UA!UAInstance)) = thisModule.overriddenIds->get(self)
    in
        if overriddenIds.oclIsUndefined() or overriddenIds->isEmpty()
        then
            self.getTypeDefinitionNode().getClassName(false)
        else
            let supertypes : Set(UA!UAType) = hierarchyRoot.getAllSuperTypes()
            in let overriddenId : TupleType(root : UA!UAType, id : UA!UAInstance) =
                overriddenIds->any(tuple | supertypes->includes(tuple.root))
            in
                if thisModule.idsWithMultipleOverriddenIds->includes(self)
                then
                    overriddenId.id.getClassName(false).prependTypeNameToInstanceDeclaration(self.getTypeDefinitionNode()).appendHierarchyRootToInstanceDeclarationName(overriddenId.root)
                else
                    overriddenId.id.getClassName(false).prependTypeNameToInstanceDeclaration(self.getTypeDefinitionNode())
                endif
        endif
    ;
```

Listing 4.5: ATL *getSuperTypeName* helper.

4.1.2 Matched Rules

Following ATL philosophy [JABK08], the greater part of the transformation is written in a declarative way, using matched rules.

TypeDefinitionNodes

The rule *UAType2CDInterface* 4.6 maps every *TypeDefinitionNode* to an *ASTCDInterface*, a CD4A AST node representing an interface.
For *TypeDefinitionNodes* that are *Interfaces* we use the standard name generated by *getClassName()* and for other *TypeDefinitionNode* we modify the CD interface's name to avoid a collision with the class that will receive the standard name.
The *ASTCDExtendUsage* object contains the name of the extended *ASTCDInterface* if the source *Interface* references another *Interface* with an inverse *HasSubType Reference*. We create no further target elements in this rule as inheritance for non-*Interface TypeDefinitionNodes* is handled on the class level and any associations are created in another rule, matching *References*.
In a short imperative section we add all created interfaces to the *ASTCDPackage* provided by a native java helper. In theory such an imperative section could be avoided for matched rules by using ATL's tracing mechanism to collect the target elements in an endpoint rule. However, due to the difficulties with EMFTVM's lazy lists, we use this approach instead.

```
rule UAType2CDInterface {
    from
        uaType : UA!UAType (
            (uaType.oclIsKindOf(UA!UAVariableType) or
            uaType.oclIsKindOf(UA!UAObjectType)) and
            thisModule.relevantNodes->get(uaType)
        )
    using {
        superType : UA!UAType = uaType.getDirectSuperType();
    }
    to
        cdInterface : CD4A!ASTCDInterface(
            Name <- uaType.getClassName(not uaType.isInterface()),
            CDExtendUsage <- extendUsage
        ),
        extendUsage : CD4A!ASTCDExtendUsage (
            Superclass <-
                if superType.oclIsUndefined()
                then
                    Sequence{}
                else
                    Sequence{
                        thisModule.UAType2SuperclassName(superType)
                    }
                endif
        )
    do {
        thisModule.cdPackage.addCDElement(cdInterface);
    }
}
```

Listing 4.6: ATL matched rule *UAType2CDInterface*.

We create an *ASTCDClass* for any non-*Interface TypeDefinitionNode* by means of the *UAType2CD-Class* rule 4.7. It extends *UAType2CDInterface*, creating additional target elements and executing additional imperative code for any source elements that match its source pattern. The source pattern constitutes a refinement of the extended rule's source pattern, matching only a subset of its matched source elements.
In contrast to the interfaces created in *UAType2CDInterface*, the created classes are more complex. Every class has a *ASTModifier* element, representing multiple UML modifiers via Boolean attributes

that are set to true if the corresponding modifier is present. We use the *abstract* modifier when mapping *TypeDefinitionNodes* that are abstract or not a *TypeDefinitionNode* we want to instantiate. The latter case consists of the *TypeDefinitionNodes* that are not instantiated independently but are needed as supertypes of one or more such *TypeDefinitionNode*.
As in the *UAType2CDInterface* rule we create an *ASTCDExtendUsage* instance to describe a possible inheritance relationship with the name of the superclass. We also create an *ASTCDInterfaceUsage* instance, which collects the names of all interfaces implemented by the class. Extended classes and implemented interfaces are easily determined by simply following the *HasSubType Reference* and all forward *HasInterface References*. Additionally, any class always implements its corresponding interface created from the same *TypeDefinitionNode* in the parent rule *UAType2CDInterface*. The names and their containing objects are obtained via the lazy rules *UAType2SuperclassName* and *UAType2InterfaceName*.
Again we use a small imperative section to collect the classes in the *ASTCDPackage*.

```
rule UAType2CDClass extends UAType2CDInterface {
    from
        uaType : UA!UAType (
            (uaType.oclIsTypeOf(UA!UAVariableType) or uaType.oclIsTypeOf(UA!UAObjectType)) and
            (not uaType.isInterface()) and
            thisModule.relevantNodes->get(uaType)
        )
    using {
        superType : UA!UAType = uaType.getDirectSuperType();
    }
    to
        cdClass : CD4A!ASTCDClass (
            Name <- uaType.getClassName(false),
            CDExtendUsage <- extendUsage,
            CDInterfaceUsage <- interfaceUsage,
            Modifier <- umlModifier
        ),
        umlModifier : CD4A!ASTModifier (
            R__abstract <- uaType.isAbstract(),
            R__public <- not (uaType.getAllSuperTypes()->including(uaType)->exists(type | type.isIndependent()))
        ),
        extendUsage : CD4A!ASTCDExtendUsage (
            Superclass <-
                if superType.oclIsUndefined()
                then
                    Sequence{}
                else
                    Sequence{thisModule.UAType2SuperclassName(superType)}
                endif
        ),
        interfaceUsage : CD4A!ASTCDInterfaceUsage (
            R__interface <- uaType.getInterfaces()->including(uaType)->collect(interface | thisModule.
UAType2InterfaceName(interface))
        )
    do {
        thisModule.cdPackage.addCDElement(cdClass);
    }
}
```

Listing 4.7: ATL matched rule *UAType2CDClass*.

InstanceDeclarations

The *UAInstanceDeclaration2Class* rule A.1 maps *InstanceDeclarations* to a hierarchy of *Classes*, provided the *InstanceDeclaration* is not part of multiple *InstanceDeclarationHierarchies* where its overridden *Node* in one is different from the one in the other *InstanceDeclarationHierarchy*. We compute the set of such *InstanceDeclarations*, as well as the set of general *InstanceDeclarations* before the matching phase and need only to check if the sets include or exclude the *Instance*.

The target pattern elements in *UAInstanceDeclaration2Class* are largely the same as in the *UAType2CDClass* rule 4.7, but in this rule the class does not implement any interface and we set the attributes of the target elements differently.
The *ASTCDClass*'s *Name* is again constructed via *getClassName()*, but we append the *TypeDefinitionNode*'s *Class Name*. Determining the name of the superclass is somewhat complicated, thus we use the helper *getSuperTypeName()*.
In the *ASTModifier* the *public* modifier is always present to identify the class as based on an *InstanceDeclaration*. We set *abstract* to true iff the *TypeDefinitionNode* of the *InstanceDeclaration*, *idType*, is abstract.
Other reasons for making a class abstract as in *UAType2CDClass* 4.7 do not apply here as we always instantiate *TypeDefinitionNodes* with their *InstanceDeclarationHierarchies*, not *InstanceDeclarations* by themselves. Making *TypeDefinitionNodes* we do not want to instantiate *abstract* is therefore sufficient.

The created *ASTCDClass* is again added to the *ASTCDPackage*. In contrast to the previous rules we utilize the imperative section to create further elements. We create an additional class for any subtype of the *InstanceDeclaration*'s type. To this end we call the recursive *CreateInstanceDeclarationSubTypes* called rule A.3 on every relevant direct subtype.

References

Another matched rule A.2, *Reference2Association*, matches every forward hierarchical *Reference* between two relevant *Nodes*, where the *TargetNode* is an *InstanceDeclaration* and the *SourceNode* is either a *TypeDefinitionNode* or a non-placeholder *InstanceDeclaration*. The target pattern consists of an *ASTCDAssociation* and several contained objects describing the association.
We set the *CDAssocType* of the *ASTCDAssociation* to an instance of *ASTCDAssocTypeAssoc*, which makes it an association as opposed to, e.g., a composition. Additionally, we set *CDAssocDir* to an *ASTCDBiDir* instance, making the *ASTCDAssociation* bidirectional.

All other important properties of the *Association* are related to either the left side or the right side, from a textual perspective. Therefore we create *ASTCDAssocLeftSide* and *ASTCDAssocRightSide* instances, with the left side always corresponding to the *SourceNode* and the right side to the *TargetNode*. They contain the cardinality, the name of the respective class, and a role name.
We construct the role names from the *BrowseNames* of the *ReferenceType*, *SourceNode* and *TargetNode*, thus ensuring overriding *References* map to associations with the same role names. For the *SourceNode*, since the *Reference* may override a *Reference* specified by the *TypeDefinitionNode*, we do not use the *BrowseName* of the *SourceNode* itself. Instead the *BrowseName* of the highest-level supertype, where either the *TypeDefinitionNode* itself or an *InstanceDeclaration* of its type contains a *Reference* of the same *ReferenceType* to an *InstanceDeclaration* with the same *BrowseName*, is

used. This way *References* of *TypeDefinitionNodes* are overridden correctly. However, at the time of writing the implementation does not support overriding of *References* with some *ReferenceType* by *References* of a subtype. Regarding uniqueness of the field names for a class, this is sufficient for the *SourceNode* as *BrowseNames* must be unique within an *InstanceDeclarationHierarchy*. For the *TargetNode*, however, we assume in our current implementation that it is not referenced by a *Node* with the same *BrowseName* as the *TargetNode*. Moreover, as mentioned in chapter 3, the current naming strategy allows for overriding of *References* only if the *ReferenceType* is identical.
The names of the respective classes are built the same way as they are for the target class in *UAType2CDClass* for *TypeDefinitionNodes* or *UAInstanceDeclaration2Class* for *InstanceDeclarations*.

The cardinalities are represented by objects of specific classes, e.g., *ASTCDCardOne* or *ASTCDCardMult*. For the left side, i.e., the side of the *TargetNode*, we always use *ASTCDCardMult* and create it as part of the target pattern. For the right side we use specific called rules depending on the *ModellingRule* of the *TargetNode*. The mapping of *ModellingRules* to cardinalities 3.2 is given in chapter 3. In order not to clutter the target pattern with convoluted if-then-else-clauses, we call the appropriate rule in the imperative section and add it to the *ASTCDAssocRightSide* object.

Additionally, as with the other matched rules, we add our top-level target element, the *ASTCDAssociation*, to the *Package*.

4.1.3 Called Rules

The called rule *CreateInstanceDeclarationSubTypes* A.3 creates a hierarchy of subclasses for an *InstanceDeclaration*, corresponding to the subtypes of the *InstanceDeclaration*'s type. For each call one specific *ASTCDClass* is created for the given *InstanceDeclaration* and its *TypeDefinitionNode idType*. The rule calls itself recursively for every direct subtype of *idType* with the respective *TypeDefinitionNode* as new *idType*.
Target elements other than *superclassName* and *interfaceUsage* do not differ significantly from those in *UAInstanceDeclaration2Class*. In *superclassName* we use the *supertypeName* argument, which the calling rule always sets to its respective class name. For *interfaceUsage* we collect all interfaces implemented by the class corresponding to *idType* (see chapter 3).

4.1.4 Lazy Rules

In CD4A-ASTs superclasses and implemented interfaces for an *ASTCDClass* or *ASTCDInterface* object are given via *ASTCDExtendUsage* and *ASTCDInterfaceUsage* objects by means of the respective class or interface names. Implementation-wise they contain an *ASTMCQualifiedType*, containing an *ASTMCQualifiedName* which then contains the actual string.
The necessary objects are created using the lazy rules *UAType2SuperclassName* and *UAType2InterfaceName* that use *getClassName()* to generate the name, thus matching the respective name in the rule *UAType2CDInterface* or *UAType2CDClass*.
UAType2InterfaceName checks if the given *TypeDefinitionNode* is an *Interface* and uses either the standard name or the modified version for a created interface accordingly. *UAType2SuperclassName*, on the other hand, always uses the standard name.

```
lazy rule UAType2InterfaceName{
        interfaceName : CD4A!ASTMCQualifiedName (
            Parts <- Sequence{
                uaType.getClassName(not uaType.isInterface())
            }
        )
  do {
      outInterface;
  }
}
```

Listing 4.8: *UAType2InterfaceName* lazy rule.

4.2 CDDiff Changes

To implement the changes to the semantic mapping discussed in section 3.2 some changes in the generated Alloy modules of *cddiff* [MRR11b] are necessary.

In order to ensure that no *InstanceDeclarations* are instantiated independently we introduce some new common signatures 4.9. *InstanceDeclarations* are mapped to public classes in our ATL transformation, thus distinguishing them from the target classes of the *independent* types (see section 4.1).
We declare a new abstract signature *InDec*, extending the existing signature *Obj*, thus creating a subset of *Obj*. While in *cddiff* a signature extending *Obj* is created for every class, in the modified version the signatures of public *Classes* extend *InDec* instead. Any two signatures extending the same signature are pairwise disjoint, therefore no atom of *Obj* can be part of both *InDec* and the signature of a non-public class.

Field names in *cddiff*'s Alloy module are defined as signatures containing exactly one atom and extending the abstract *FName* signature. Associations are then navigated via the *get* relation. For a bidirectional association there are two signatures, one from the left role name which is a field for the right class and analogously one from the right role name, which is a field for the left class. Here we introduce a new abstract signature *RFName*, which is extended by the signatures based on the left role name of an association. With these signatures we can navigate associations in what corresponds to the backward direction of the respective OPC UA *References*.

```
abstract sig Obj { get: FName -> {Obj + Val + EnumVal} }
abstract sig FName {}
abstract sig InDec extends Obj {}
abstract sig RFName extends FName {}
```

Listing 4.9: Additional signatures in generated Alloy modules.

A single new fact, *InDecHierarchy* using a simple helper function (see listing 4.10), achieves the desired effect of no independently instantiated *InstanceDeclarations*. The function *parentRel* uses the *RFName* signature to return a binary relation, mapping every object to the objects reachable by navigating one link from right to left. In OPC UA terms it maps a *Node* to the *Nodes* that reference it by means of a forward hierarchical *Reference*. Alloy includes two transitive closure operators, which the *InDecHierarchy* fact uses, thus needing this binary relation. The relation is described as a function below.

$$\text{parentRel} : Obj \to \mathcal{P}(Obj),\ t \mapsto \{\, s \mid \exists f \in RFName\ :\ (t, f, s) \in get\}$$

In the *InDecHierarchy* fact we use the transitive closure of *parentRel* to ensure that from every *InDec* atom an atom in *Obj* \ *InDec* is reachable following the links represented by the *parentRel* relation. In OPC UA terms, any *Instance* based on an *InstanceDeclaration* is directly or indirectly referenced by an *Instance* based on a *TypeDefinitionNode* through forward hierarchical *References*. Therefore, *Instances* of an *InstanceDeclaration* exist only as parts of mirrored *InstanceDeclarationHierarchies*.
Note that while we chose the non-reflexive transitive closure, it is irrelevant which option is used, since the identity gives us an $i \in InDec$ and $i \in InDec \implies i \notin Obj \setminus InDec$.

```
//Additional Fact for UADiff. Ensures that InstanceDeclarations are only instantiated as Part of a hierarchy.
fact InDecInHierarchy {
 all i: InDec | some o: (Obj-InDec) | o in (i.^parentRel)
}
fun parentRel: Obj -> Obj {
 {t:Obj, s:Obj | s in t.get[RFName]}
}
```

Listing 4.10: Additional fact and function in generated Alloy modules.

The second change to the semantic mapping requires every pair of objects to either have links corresponding to all defined or inherited associations in the same direction between their respective classes or none of them. To reflect this, we add additional constraints to the top-level predicate of the respective CDs. For classes with multiple associations in the same direction to the same class we add the constraint, that for all atoms of the corresponding signature, the sets of objects returned by the *get* relation must be pairwise equal each respective *FName*.
This is demonstrated in listing 4.11 for an example class *SomeClass* containing three fields, all corresponding to some associations with one class on the right side and *SomeClass* on the left side.

```
  all o: SomeClass | (o.get[field1]=o.get[field2]) && (o.get[field2]=o.get[field3])
```

Listing 4.11: Additional condition for references between the same *Nodes*.

4.3 Demonstrator

Using the ATL transformations and the modified *cddiff*, we implemented a demonstrator, producing textual object model witnesses as semantic difference of two OPC UA models. As additional input the demonstrator takes an upper bound *k* for the number of objects in the witnesses and the number of witnesses to be generated. Both arguments are simply passed on to *cddiff*. In this section we describe the demonstrator regarding its structure and behaviour.

4.3.1 Structure

We implemented the demonstrator as a Java application, which is the most practical choice as all used tools and frameworks are also written in Java. The overall structure is visualized as a component diagram in figure B.1, containing the following components:

UADiff is the tool's main component, launching the ATL transformations, invoking *cddiff* and providing a rudimentary command-line interface (CLI). Additionally it provides a Java class as helper for EMFTVM, which we decided to omit in the diagram, as we consider it more of a workaround to provide additional information to the transformation.

EMF refers to different packages of the Eclipse Modeling Framework, which provides the facilities to process the models and execute transformations with EMFTVM. See chapter 2 for a more detailed description.

CD4A-EMF is the MC-generated code for the CD4A language with EMF-compatible AST classes, due to an alternate workflow script in the generation process [HKR21]. The AST classes form the metamodel for CD4A. The generated *EPackages* lack the corresponding *EFactories* and contain some faults preventing immediate use in ATL transformations. Therefore we made some handwritten additions to this component.

UA-EMF refers to the classes representing the Ecore-based metamodel of OPC UA and the corresponding infrastructure, e.g., *EFactories*. It is generated from the XSD for *NodeSets* given in [OPC21c]. Contrary to *CD4A-EMF* there are no handwritten parts in this code.

EMFTVM is a VM for executing ATL transformations on EMF models [WTCJ11]. It provides a Java API including a compiler from ATL to EMFTVM's bytecode.

CDDiff is the implementation of *cddiff* [MRR11b] with some modifications to better reflect OPC UA semantics. It transforms two CD4A diagrams to an Alloy module and runs the Alloy Analyzer [JSS00] on it. In the state of the demonstrator at the time of writing it is also responsible for the final output of the witnesses, which are textual object model structures.

4.3.2 Workflow and Data Representation

The high-level workflow as depicted in figure B.2 is relatively straightforward, as we mostly use existing tools or execute the transformations, which constitute the core of this work and are already explained in detail in other sections.

As first step, the transformation is modified and then compiled. We modify the header of the ATL module to account for varying numbers of source models, as all OPC UA models except the base information model consist of at least two *NodeSets*, which are registered in the execution environment of EMFTVM as separate input models.

Then comes the transformation initialization phase, registering metamodels and compiled transformations with EMFTVM, reading the input models from their files and launching two threads for parallel execution of the transformation.

Once both transformations are finished, the target CDs are used as input for the modified CDDiff, which then outputs the diff witnesses.

Figure B.3 shows the different representations of the processed models and the transformations between them. As the OPC UA metamodel is based on an XSD file, EMF can read any XML file conforming to the XSD [SBMP11], obtaining a representation as instances of the metamodel classes.
By executing the ATL transformations on the models with EMFTVM we receive CD4A models represented by instances of CD4A's AST classes.
These models are then transformed to a single Alloy module by *cddiff*. For this module the Alloy Analyzer outputs textual Alloy solutions [JSS00]. These solutions are transformed back to a textual representation of object models, the witnesses.

5 Evaluation

The evaluation of our *uadiff* demonstrator is twofold. First we run *uadiff* on some small example *NodeSets*, which we specifically created to assess that our implementation works as intended. This is of course not comparable to thorough testing on a larger scale, but such an approach would not be feasible within the scope of this work.
Second we evaluate the performance of our *uadiff* demonstrator on *NodeSets* used in practice that were published by the OPC Foundation [OPC22].

5.1 Validity on Simple Example

We chose to adapt the apes and bananas CD to OPC UA in two slightly different versions, A_1 5.1 and A_2 5.2. While many possible cases in the ATL transformation are obviously not covered by this simple example, we can observe two key mechanisms of *uadiff*:

- *InstanceDeclarations* are not instantiated unless they are part of a mirrored *InstanceDeclarationHierarchy*.
- *InstanceDeclarations* override *InstanceDeclarations* with the same *BrowsePath* in a supertype's *InstanceDeclarationHierarchy*.

The models include *ObjectTypes* Animal, Ape, Fruit and Banana, where Ape is a subtype of Animal and Banana is a subtype of Fruit. Animal references an *InstanceDeclaration* FruitForAnimal with a forward hierachical *Reference*. In both depicted models, the FruitForAnimal *InstanceDeclaration* is overridden in the *InstanceDeclarationHierarchy* of the Ape *ObjectType*, such that it is of type Banana. In 5.1 the *ModellingRule* of this overriding *InstanceDeclaration* remains *Optional*, while in 5.2 it is *Mandatory*.
Informally, both models specify the *ObjectTypes* Animal and Fruit, where any Animal may eat a Fruit. Since *Nodes* can be the target of arbitrary many forward hierarchical *References*, multiple Animals can eat a single Fruit. For Apes the reference is overridden, therefore Apes eat only Bananas. Contrary to the CD this example is based on, the Bananas eaten by Apes are a distinct subclass of the Bananas eaten by Animals, thus enabling Animals that are not Apes to eat Bananas.

We run *uadiff*(A_1,A_2) with Animal and Ape as independent *Nodes* and Fruit and Banana as included Types (see section 4.1.1). All generated witnesses contain at least one Ape without any links. When further objects are part of the witness, they contain no Fruit or Banana that is not eaten by at least one Animal or Ape and Apes eat only Bananas. An example of such a witness is the listing 5.1.

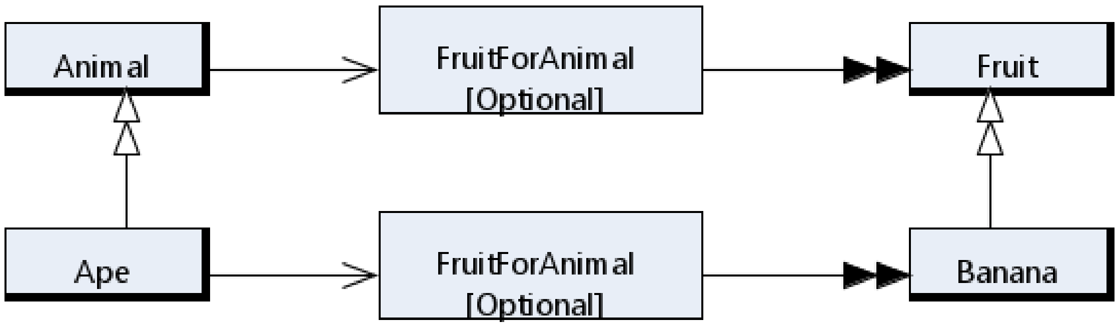


Figure 5.1: A_1, an OPC UA adaptation of 3.1 featuring *Optional* Bananas.

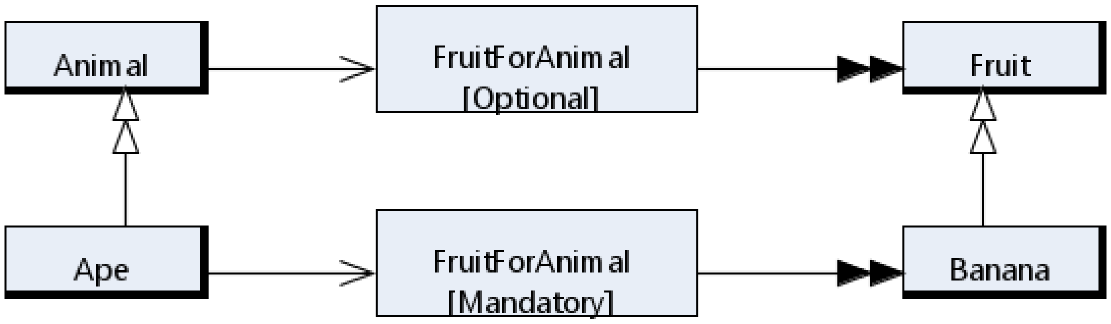


Figure 5.2: A_2, an OPC UA adaptation of 3.1 featuring *Mandatory* Bananas.

5.2 Performance on Real-World NodeSets

We evaluate the performance of *uadiff* based on the execution time and the number of created classes and interfaces in relation to the number of relevant (see section 4.1.1) *Nodes* in the input model. The time measurements give an easily understandable metric for the performance of the ATL transformations executed by EMFTVM on the selected input. While still limited by the sample size, the number of generated classes provide an estimate of how the number of relevant *Nodes* affects the input size for *cddiff*, which is the dominant factor for execution time when working on large portions of the usually large *NodeSets*.

5.2.1 Setup and Choice of Input

For the evaluation *uadiff* was executed on a desktop computer using Windows 10, an Intel quad-core CPU at $3.4GHz$, and $16GB$ RAM. The application was run 50 times from a batch file for each input since subsequent executions in a loop within the Java application tend to be faster and many consecutive runs on the exact same input are unlikely to happen in practice. The elapsed time was measured within the Java application and written to a CSV file. It includes the time for compiling and initializing the transformation. As the number of relevant *Nodes* and generated classes do not change between runs on the same input, the numbers were simply printed as console output. The *NodeSets* include some *NodeSets* from a GitHub repository published by the OPC Foundation [OPC22], the base information model from [OPC21c] and our own basic example from the previous

```
objectdiagram od17 {
  UABanana_UAFruitForAnimal_ns1i50020:UABanana_UAFruitForAnimal_ns1i5002 {
  };
  UAApe0:UAApe {
  };
  UAApe1:UAApe {
  };
  link UABanana_UAFruitForAnimal_ns1i50020 -> (source_UAEats_Animal_2_FruitForAnimal) UAApe1;
  link UAApe1 -> (target_UAEats_Animal_2_FruitForAnimal) UABanana_UAFruitForAnimal_ns1i50020;
}
```

Listing 5.1: Witness produced by *uadiff*A_1, A_2

section 5.1. The choice of *NodeSets* was limited to such *NodeSets* that depend only on the base information model. *NodeIds* in *References* to *Nodes* in other *NodeSets* need to be mapped to their counterpart in the containing *NodeSet*, which usually has a different *NameSpaceIndex* component. At the time of writing our ATL implementation has no way of handling this translation between *NodeSets*. The choice of independent *TypeDefinitionNodes* was not subject to any criteria other than to include varying numbers of relevant *Nodes*

5.2.2 Results

The results for generated classes and interfaces can be found in table 5.1.
Note that number of interfaces is exactly the number of relevant *TypeDefinitionNodes*. The same is true for associations and relevant forward hierarchical *References*, which is why we chose to exclude them from the table.
How many classes are generated in relation to the number of relevant *Nodes*, on the other hand, varies greatly depending on the *NodeSet* and selected *TypeDefinitionNode*. Also note that the *ObjectTypes* in our example *NodeSet* A_1 are subtypes of *BaseObjectType*, which brings the relevant *Nodes* to 7.

NodeSet	**Independent Type**	**Relevant Types**	**Relevant Instances**	**Classes**	$\frac{\lvert\text{Classes}\rvert+\lvert\text{Interfaces}\rvert}{\lvert\text{Nodes}\rvert}$
Example A_1 (5.1)	Animal	5	2	8	1.86
CommercialKitch-enEquipment	IceMachineParame-terType	9	20	41	1.72
MachineVision	RecipeManage-mentType	19	46	83	1.57
PackML	PackMLBaseOb-jectType	20	108	487	3.96
DI	TopologyElement-Type	21	97	167	1.59
OPC UA Base Infor-mation Model	BaseObjectType	262	1480	18454	10.74

Table 5.1: *ModellingRule* of the *Reference*'s *TargetNode* in OPC UA mapped to UML multiplicities

The average execution times for the transformations within the corresponding *uadiff* runs are shown in table 5.2. For smaller sets of relevant *Nodes*, the initialization phase and some helpers that are always executed for the whole *NodeSet* are the dominant factors regarding transformation performance. The time is always for two parallel executions, as every run of *uadiff* includes two transformations of two, usually similar, OPC UA models.

NodeSets	**Relevant Types**	**Time (s)**
Example A_1 (5.1)	Animal	5
CommercialKitchenEquipment	IceMachineParameterType	10.9
MachineVision	RecipeManagementType	11.0
PackML	PackMLBaseObjectType	10.9
DI	TopologyElementType	10.2
OPC UA Base Information Model	BaseObjectType	62.2

Table 5.2: Execution time for two parallel transformations on the same *NodeSet*. The values are averaged over consecutive 50 executions of *uadiff*.

For detailed results on the performance of *cddiff* see [MRR11b]. However, we can state that *cddiff* runs relatively fast for sensible input sizes provided the maximum number of objects in a witness, k, is not set too high and even much faster than our transformation when using small sets of relevant *Nodes*.
In our Animals example 5.1 we had *cddiff* execution times between 0.8 and 1.2 seconds for $k = 5$ to $k = 10$, while for the *RecipeManagementType* in the MachineVision *NodeSet* these times are already up to 50 seconds and 5 minutes, respectively.

5.3 Discussion

The results show that our implementation of the *uadiff* operator is sufficiently fast to be used in practice on reasonably sized subsets of OPC UA information models. On large inputs the performance declines notably, as the produced CDs for such inputs are multiple times larger regarding the number of contained elements. However, exceedingly large models are ill-suited for enumerative semantic differencing approaches not just from a performance perspective.
The difference would either consist of very large witnesses or of small witnesses that in most cases could have been produced when examining a much smaller subset of the model elements. In the former case it would become increasingly difficult for the user to determine the reason why a model is a witness.
Considering the time for initialization of EMFTVM the time for execution of *uadiff* cannot be improved beyond a couple of seconds, even for smaller models. Therefore we consider the performance of our demonstrator to be within an acceptable range.
Furthermore, we demonstrated on a simple example that our implementation works as described in chapters 3 and 4. While more testing needs to be done before any practical use, in combination with the performance evaluation it shows basic viability of our demonstrator.

5.4 Threats to Validity

Apart from the ones in [OPC22], publically available *NodeSets* that are used in industry are scarce. The selection of used input models is further limited by the capabilities of our demonstrator, which, at the time of writing, cannot handle translation of *NodeIds* in *References* between different *NodeSets*. Additionaly, we took into account the size of samples and how many *TypeDefinitionNodes* or *InstanceDeclarations* were included in the relevant subset, but did not take the respective subset into account. Especially the depth of subtype hierarchies and the number of overrides influence the number of generated classes. This is somewhat mitigated by the fact, that, in order to produce useful witnesses, *uadiff* should be applied on relatively small subsets of *NodeSets* anyway. Lastly we assume that our modified version of *cddiff* performs similar to the original. Any additional constraints we introduced to the Alloy modules are rather simple and we add only two abstract signatures. Therefore, we believe a similar performance to be a reasonable assumption, but did not verify it directly. It is clear, however, that the modified version still runs reasonably fast on CDs generated from appropriately small inputs to *uadiff*.

6 Conclusion

We conclude our thesis with brief summary and discussion. This includes the benefits of this work, as well as its limitations. Lastly, we give an overview of possible future research.

6.1 Summary

In this work we presented the, to our knowledge, first semantic differencing operator for OPC UA. We proposed adapted semantics for OPC UA models, imposing constraints on instantiation such that the models describe instances in a *complete* [Rum11] way. To achieve this, we designed an ATL transformation from OPC UA to UML CDs, or to be more precise, UML/P [Rum11] CDs represented in the language CD4A [HKR21]. Due to OPC UA not separating different levels of abstraction in its models, finding an appropriate mapping represents a challenge. The concepts of *InstanceDeclarations* and overriding of the same cannot be easily emulated using CDs.
However, using the transformation coupled with a slightly adapted *cddiff* [MRR11b] we were able to construct a demonstrator, *uadiff*, which produces witness object diagram structures representing instances of OPC UA models.
We were able to confirm on a simple example *NodeSet* of our own design, including several key features, e.g., overriding of *InstanceDeclarations*, that *uadiff* works as intended. Performance evaluation on industry-relevant *NodeSets* revealed that *uadiff* is well suited for smaller models, which constitute ideal targets for a semantic differencing tool. From given input *NodeSet* and *TypeDefinitionNodes* of interest, we generate subsets of the typically large input *NodeSets* to achieve such smaller models. We learned that our transformation continues to work in acceptable time frames even for large inputs, whereas this is not the case for the overall tool.

6.2 Benefits

Designers of OPC UA models can use *uadiff* to gain insights on how the possible instances of their models change between versions. The tool is intended to supplement usual syntactic differencing solutions that are already in use. Combined with such a syntactic differencing tool it can simplify the process of identifying changes and the corresponding effects. This could be especially helpful for developers without in-depth knowledge of OPC UA.

Additionally we hope that tools with a complete interpretation of OPC UA models might lead to higher quality models. While it is perfectly possible in OPC UA to specify very general types and add *References* to their instances as needed, this could lead to a very heterogeneous landscape of models even for similar applications. Furthermore, incomplete type specifications are more difficult to understand for users. However, the contribution of a tool to such a goal depends on the actual use and acceptance, which might depend on the willingness to adopt a stricter modeling approach in the first place.

6.3 Limitations

There are a number of limitations to our operator and its implementation. Some are inherent to OPC UA models, mostly due to the lack of separation between type definitions and instances.

A complete interpretation of OPC UA models cannot include all the possibilities of the standard interpretation, e.g., *Instances* of different non-overriding *InstanceDeclaration* with the same *Type-DefinitionNode* and *BrowseName* cannot be used interchangeably. While our implementation does not permit this, even if the *References* defined by the *InstanceDeclaration* were identical, a more permissive operator still could not allow it with different *References* without giving up on the complete interpretation to a degree.

Regarding our operator and interpretation of OPC UA, the semantics are not quite as well-defined as, e.g., for *cddiff*. Due to our modification of *cddiff* we alter the semantics of UML/P without formally defining them. Moreover, the ATL module contains imperative code and some Java helpers, which are less concise then a purely declarative approach.

For our implementation approach, overriding *Reference* of some type A with a *Reference* of some subtype of A will be difficult to implement, as the overriding mechanism depends on the overriding association having the same role names, which include the *ReferenceType*'s *BrowseName*.
Also the approach is inherently unsuitable for very large input models that cannot be split into smaller parts, due to increasingly large alloy modules and thus Boolean formulas generated.
We mentioned further limitations of our implementation throughout chapters 4 and 5, but these are mostly not inherent limitations of our approach, but rather features we did not implement due to time limitations.

6.4 Lessons Learned

Trying to develop a tool such as semantic differencing for OPC UA, which despite some obvious similarities differs significantly from UML class diagrams and the object oriented systems they represent, shows the importance of a clear separation between different levels of abstraction. From a user or even model designer perspective, the impact might seem relatively small at first, but when considering the semantics of such models in-depth, problems arise that complicate, e.g., the development of tools considerably.

6.5 Future Work

The tool in its current form serves only as proof of concept and there are a number of ways it could be improved. A transformation from the textual object models back to OPC UA *Instances* and maybe even a visual representation would make the results easier to understand. Completing the OPC UA to CD4A transformation is another possibility, enabling the use of *uadiff* on more complicated models, e.g., ones that include *InstanceDeclaration* overriding multiple other *InstanceDeclarations*.

Additionally, many future research challenges that were proposed in [MRR12] are worth pursuing in the context of OPC UA. Especially integration with syntactic differencing to localize changes and summarization to avoid showing many similar witnesses seem promising for *uadiff*.

Lastly, another operator could be developed avoiding a transformation to CDs and the restrictions that come with it.

Bibliography

[AGH05] K. Arnold, J. Gosling, D. Holmes. *The Java programming language*. Addison Wesley Professional, 2005 (cit. on p. 14).

[AP03] M. Alanen, I. Porres. "Difference and union of models". In: *International Conference on the Unified Modeling Language*. Springer. 2003, pp. 2–17 (cit. on p. 15).

[Béz05] J. Bézivin. "Model driven engineering: An emerging technical space". In: *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer. 2005, pp. 36–64 (cit. on p. 6).

[CGR14] M. V. Cengarle, H. Grönninger, B. Rumpe. "System model semantics of class diagrams". In: *arXiv preprint arXiv:1409.6635* (2014) (cit. on p. 19).

[CK07] T. Cleenewerck, I. Kurtev. "Separation of concerns in translational semantics for DSLs in model engineering". In: *Proceedings of the 2007 ACM symposium on applied computing*. 2007, pp. 985–992 (cit. on p. 19).

[CS14] S. Chacon, B. Straub. *Pro git*. Springer Nature, 2014 (cit. on p. 14).

[FALW14] U. Fahrenberg, M. Acher, A. Legay, A. Wąsowski. "Sound merging and differencing for class diagrams". In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2014, pp. 63–78 (cit. on pp. 16, 17).

[FLW11] U. Fahrenberg, A. Legay, A. Wąsowski. "Vision paper: Make a difference!(semantically)". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2011, pp. 490–500 (cit. on p. 17).

[HKR21] K. Hölldobler, O. Kautz, B. Rumpe. *MontiCore Language Workbench and Library Handbook: Edition 2021*. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021. ISBN: 978-3-8440-8010-0. URL: http://www.monticore.de/handbook.pdf (cit. on pp. 14, 18, 40, 49).

[HR04] D. Harel, B. Rumpe. "Meaningful modeling: what's the semantics ofßemantics"?" In: *Computer* 37.10 (2004), pp. 64–72 (cit. on pp. 16, 19).

[JABK08] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev. "ATL: A model transformation tool". In: *Science of computer programming* 72.1-2 (2008), pp. 31–39 (cit. on pp. 2, 6, 33).

[Jac12] D. Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012 (cit. on p. 16).

[Jac19] D. Jackson. "Alloy: a language and tool for exploring software designs". In: *Communications of the ACM* 62.9 (2019), pp. 66–76 (cit. on p. 5).

[JSS00] D. Jackson, I. Schechter, H. Shlyahter. "Alcoa: the alloy constraint analyzer". In: *Proceedings of the 22nd international conference on Software engineering*. 2000, pp. 730–733 (cit. on pp. 5, 16, 40, 41).

[KRV10] H. Krahn, B. Rumpe, S. Völkel. “MontiCore: a framework for compositional development of domain specific languages”. In: *International journal on software tools for technology transfer* 12.5 (2010), pp. 353–372 (cit. on p. 14).

[LKYO17] B. Lee, D.-K. Kim, H. Yang, S. Oh. “Model transformation between OPC UA and UML”. In: *Computer Standards & Interfaces* 50 (2017), pp. 236–250 (cit. on p. 17).

[LM06] S.-H. Leitner, W. Mahnke. “OPC UA–service-oriented architecture for industrial applications”. In: *ABB Corporate Research Center* 48.61-66 (2006), p. 22 (cit. on p. 1).

[MRR10] S. Maoz, J. O. Ringert, B. Rumpe. “A manifesto for semantic model differencing”. In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2010, pp. 194–203 (cit. on pp. 1, 2, 15).

[MRR11a] S. Maoz, J. O. Ringert, B. Rumpe. “ADDiff: semantic differencing for activity diagrams”. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011, pp. 179–189 (cit. on pp. 1, 17).

[MRR11b] S. Maoz, J. O. Ringert, B. Rumpe. “CDDiff: Semantic differencing for class diagrams”. In: *European Conference on Object-Oriented Programming*. Springer. 2011, pp. 230–254 (cit. on pp. 1, 2, 14–17, 19, 26, 27, 38, 40, 46, 49).

[MRR12] S. Maoz, J. Ringert, B. Rumpe. “An Interim Summary on Semantic Model Differencing”. In: *Softwaretechnik-Trends* 32.4 (2012), pp. 44–46. DOI: 10.1007/BF03323524 (cit. on pp. 1, 17, 51).

[MRR14] S. Maoz, J. O. Ringert, B. Rumpe. “Summarizing semantic model differences”. In: *arXiv preprint arXiv:1409.2307* (2014) (cit. on p. 17).

[MV06] T. Mens, P. Van Gorp. “A taxonomy of model transformation”. In: *Electronic notes in theoretical computer science* 152 (2006), pp. 125–142 (cit. on p. 6).

[Obj06] Object Management Group (OMG). *Meta-Object Facility (MOF) Specification, Version 2.0*. OMG Document Number formal/2006-01-01 (http://www.omg.org/spec/MOF/2.0). 2006 (cit. on p. 6).

[OPC20] OPC Foundation. *OPC Unified Architecture Part 5: Information Model*. Release 1.05.00. OPC Foundation. Oct. 2020 (cit. on p. 1).

[OPC21a] OPC Foundation. *OPC UA Companion Specification Template*. Version 1.01.14. OPC Foundation. Nov. 2021 (cit. on pp. 65, 66).

[OPC21b] OPC Foundation. *OPC Unified Architecture Part 3: Address Space Model*. Release 1.05.00. OPC Foundation. Oct. 2021 (cit. on pp. 1, 5, 8, 10, 11, 13, 19, 24, 25, 65, 66).

[OPC21c] OPC Foundation. *OPC Unified Architecture Part 5: Mappings*. Release 1.05.00. OPC Foundation. Oct. 2021 (cit. on pp. 19, 40, 44).

[OPC22] OPC Foundation. *Project Title*. https://github.com/OPCFoundation/UA-Nodeset. 2022 (cit. on pp. 43, 44, 47).

[OWK03] D. Ohst, M. Welle, U. Kelter. “Differences between versions of UML diagrams”. In: *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. 2003, pp. 227–236 (cit. on p. 15).

[PWFW18] F. Pauker, S. Wolny, S. M. Fallah, M. Wimmer. "UML2OPC-UATransforming UML Class Diagrams to OPC UA Information Models". In: *Procedia CIRP* 67 (2018), pp. 128–133 (cit. on p. 17).

[RPL13] S. Rohjans, K. Piech, S. Lehnhoff. "UML-based modeling of OPC UA address spaces for power systems". In: *2013 IEEE International Workshop on Inteligent Energy Systems (IWIES)*. IEEE. 2013, pp. 209–214 (cit. on p. 17).

[Rum11] B. Rumpe. *Modellierung mit UML*. Springer Berlin Heidelberg, 2011 (cit. on pp. 2, 14, 16, 18, 26, 49).

[RY07] T. Rothwell, J. Youngman. "The gnu c reference manual". In: *Free Software Foundation, Inc* (2007), p. 86 (cit. on p. 14).

[SB13] M. H. Schwarz, J. Börcsök. "A survey on OPC and OPC-UA: About the standard, developments and investigations". In: *2013 XXIV International Conference on Information, Communication and Automation Technologies (ICAT)*. IEEE. 2013, pp. 1–6 (cit. on p. 1).

[SBMP11] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro. *Eclipse modeling framework: EMF*. Englisch. 2. ed., rev. and updated, 2. printing. The eclipse series. UB Vaihingen. Upper Saddle River, NJ ; Munich [u.a.]: Addison-Wesley, 2011, XXIX, 704 Seiten. ISBN: 0321331885 (cit. on pp. 6, 14, 18, 41).

[WHR13] J. Whittle, J. Hutchinson, M. Rouncefield. "The state of practice in model-driven engineering". In: *IEEE software* 31.3 (2013), pp. 79–85 (cit. on p. 6).

[WTCJ11] D. Wagelaar, M. Tisi, J. Cabot, F. Jouault. "Towards a general composition semantics for rule-based model transformation". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2011, pp. 623–637 (cit. on pp. 6, 40).

All links were last followed on June 20, 2022.

A Listings

```
rule InstanceDeclaration2Class {
    from
        uaInstance : UA!UAInstance (
            thisModule.instanceDeclarations->includes(uaInstance) and
            thisModule.idsWithMultipleOverriddenIds->excludes(uaInstance) and
            thisModule.relevantNodes->get(uaInstance)
        )
    using {
        idType : UA!UAType = uaInstance.getTypeDefinitionNode();
        hierarchyRoot : UA!UAType = thisModule.hierarchyRoots->get(uaInstance)->any(type | true);
        overriddenID : TupleType(root : UA!UAType, Id : UA!UAInstance) =
            let overriddenIds : Set(TupleType(root : UA!UAType, Id : UA!UAInstance)) = thisModule.overriddenIds->get(
uaInstance)
            in
                if overriddenIds.oclIsUndefined()
                then
                    OclUndefined
                else
                    OclUndefined
                endif
            ;
        directSubtypes : Set(UA!UAType) =
            let subtypes : UA!UAType = thisModule.directSubtypes->get(idType)
            in
                if subtypes.oclIsUndefined()
                then
                    Set{}
                else
                    subtypes->select(subtype | thisModule.relevantNodes->get(subtype))
                endif
            ;
        className : String = uaInstance.getClassName(false).prependTypeNameToInstanceDeclaration(idType);
    }
    to
        cdClass : CD4A!ASTCDClass (
            Name <- className,
            CDExtendUsage <- extendUsage,
            Modifier <- umlModifier
        ),
        umlModifier : CD4A!ASTModifier (
            --if the instance declaration is of abstract type, any instance must be of a concrete subtype
            R__abstract <- idType.isAbstract,
            R__public <- true
        ),
        extendUsage : CD4A!ASTCDExtendUsage (
            Superclass <- Sequence{superclass}
        ),
        superclass : CD4A!ASTMCQualifiedType (
            MCQualifiedName <- superclassName
        ),
        superclassName : CD4A!ASTMCQualifiedName (
            Parts <- Sequence{ uaInstance.getSuperTypeName(hierarchyRoot) }
        ),
        interfaceUsage : CD4A!ASTCDInterfaceUsage (
        )
    do {
        for (subtype in directSubtypes) {
            thisModule.CreateInstanceDeclarationSubTypes(uaInstance, className, hierarchyRoot, subtype);
        }
        thisModule.cdPackage.addCDElement(cdClass);
    }
}
```

Listing A.1: ATL matched rule *InstanceDeclaration2Class*.

```
rule Reference2Association {
    from
        uaRef : UA!Reference (
        )
    using {
        leftName : String =
            if sourceNode.oclIsKindOf(UA!UAType)
            then
                sourceNode.getClassName(not sourceNode.isInterface()) --Association left side is the interface created
 from the type
            else
                sourceNode.getClassName(false).prependTypeNameToInstanceDeclaration(sourceNode.getTypeDefinitionNode()
)
            endif;
        rightName : String =
                targetNode.getClassName(false).prependTypeNameToInstanceDeclaration(targetNode.getTypeDefinitionNode()
)
            ;
        refTypeName : String = thisModule.getNodeById(uaRef.referenceType).getClassName(false);
        sourceType : UA!UAType =
            if sourceNode.oclIsKindOf(UA!UAType)
            then
                sourceNode
            else
                sourceNode.getTypeDefinitionNode()
            endif;
        --allows overriding of type-defined references, but is problematic when two distinct IDs of the same type
reference the same ID
        sourceNameForRole : String = sourceType.findTopLevelTypeWithRef(thisModule.getNodeById(uaRef.referenceType),
targetNode.browseName).browseName;
    }
    to
        cdAssociation : CD4A!ASTCDAssociation (
            CDAssocType <- cdAssocType,
            CDAssocDir <- cdAssocDir,
            Left <- cdAssocLeftSide,
            Right <- cdAssocRightSide
        ),
        cdAssocType : CD4A!ASTCDAssocTypeAssoc,
        cdAssocDir : CD4A!ASTCDBiDir,
        --left side: source node
        cdAssocLeftSide : CD4A!ASTCDAssocLeftSide (
            CDCardinality <- cdLeftCardinality,
            MCQualifiedType <- cdLeftType,
            CDRole <- cdLeftRole
        ),
        cdLeftCardinality : CD4A!ASTCDCardMult,
        cdLeftRole : CD4A!ASTCDRole (
            Name <- 'source_' + refTypeName + '_' + sourceNameForRole.removeReservedChars()
                + '_2_' + targetNode.browseName.removeReservedChars()
        ),
        -- right side: target node
        cdAssocRightSide : CD4A!ASTCDAssocRightSide (
            MCQualifiedType <- cdRightType,
            CDRole <- cdRightRole
        ),
        cdRightRole : CD4A!ASTCDRole (
            Name <- 'target_' + refTypeName + '_' + sourceNameForRole.removeReservedChars()
                + '_2_' + targetNode.browseName.removeReservedChars()
        )
}
```

Listing A.2: ATL matched rule *Reference2Association*.

```
rule CreateInstanceDeclarationSubTypes(instanceDeclaration : UA!UAInstance, supertypeName : String, hierarchyRoot : UA
!UAType, idType : UA!UAType){
  using {
      directSubtypes : Set(UA!UAType) =
            let subtypes : UA!UAType = thisModule.directSubtypes->get(idType)
                in
                    if subtypes.oclIsUndefined()
                    then
                        Set{}
                    else
                        subtypes->select(subtype | thisModule.relevantNodes->get(subtype))
                    endif
            ;
      className : String =
            let baseName : String = instanceDeclaration.getClassName(false).prependTypeNameToInstanceDeclaration(
idType)
                in
                    if thisModule.idsWithMultipleOverriddenIds->includes(instanceDeclaration)
                    then
                        baseName.appendHierarchyRootToInstanceDeclarationName(hierarchyRoot)
                    else
                        baseName
                    endif
            ;
  }
  to
        cdClass : CD4A!ASTCDClass(
            Name <- className,
            CDInterfaceUsage <- interfaceUsage,
            CDExtendUsage <- extendUsage,
            Modifier <- umlModifier
        ),
        umlModifier : CD4A!ASTModifier (
            --if the instance declaration is of abstract type, any instance must be of a concrete subtype
            R__abstract <- idType.isAbstract,
            R__public <- true
        ),
        extendUsage : CD4A!ASTCDExtendUsage (
            Superclass <- Sequence{superclass}
        ),
        superclass : CD4A!ASTMCQualifiedType (
            MCQualifiedName <- superclassName
        ),
        superclassName : CD4A!ASTMCQualifiedName (
            Parts <- Sequence{
                supertypeName
            }
        ),
        interfaceUsage : CD4A!ASTCDInterfaceUsage (
            R__interface <- idType.getInterfaces()->including(idType)->collect(interface | thisModule.
UAType2InterfaceName(interface))
        )
  do {
      for (subtype in directSubtypes) {
            thisModule.CreateInstanceDeclarationSubTypes(instanceDeclaration, className, hierarchyRoot, subtype);
        }
      thisModule.cdPackage.addCDElement(cdClass);
  }
}
```

Listing A.3: ATL called rule *CreateInstanceDeclarationSubTypes*

B Figures

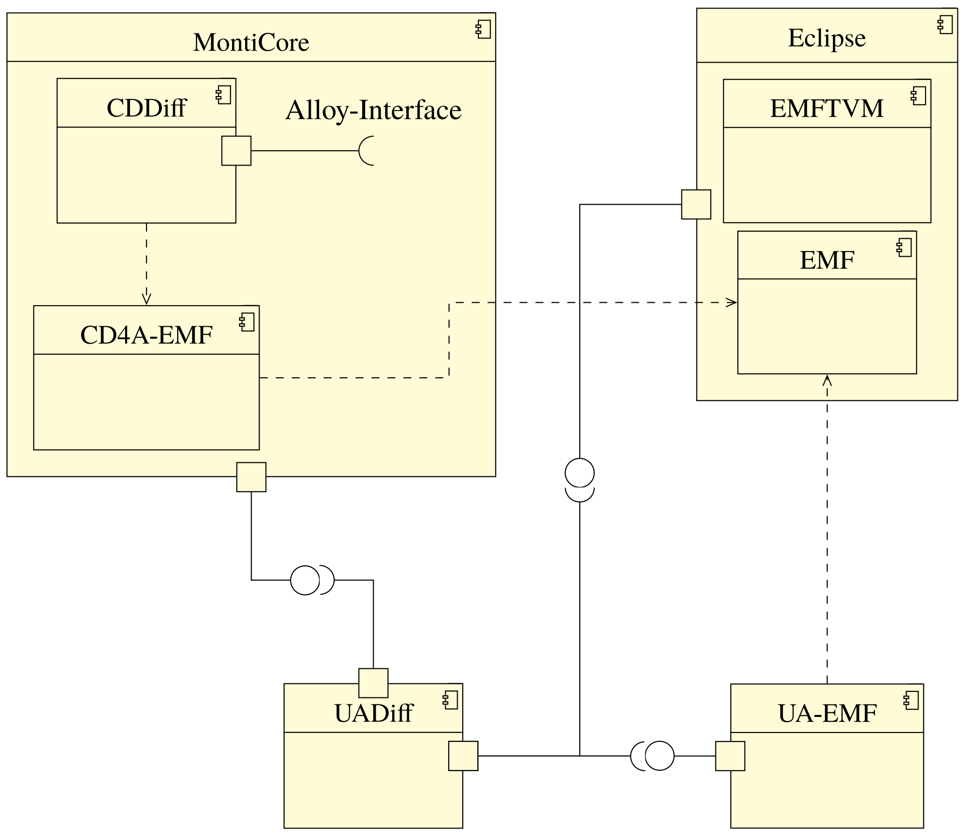


Figure B.1: Structure of the *uadiff* Application.

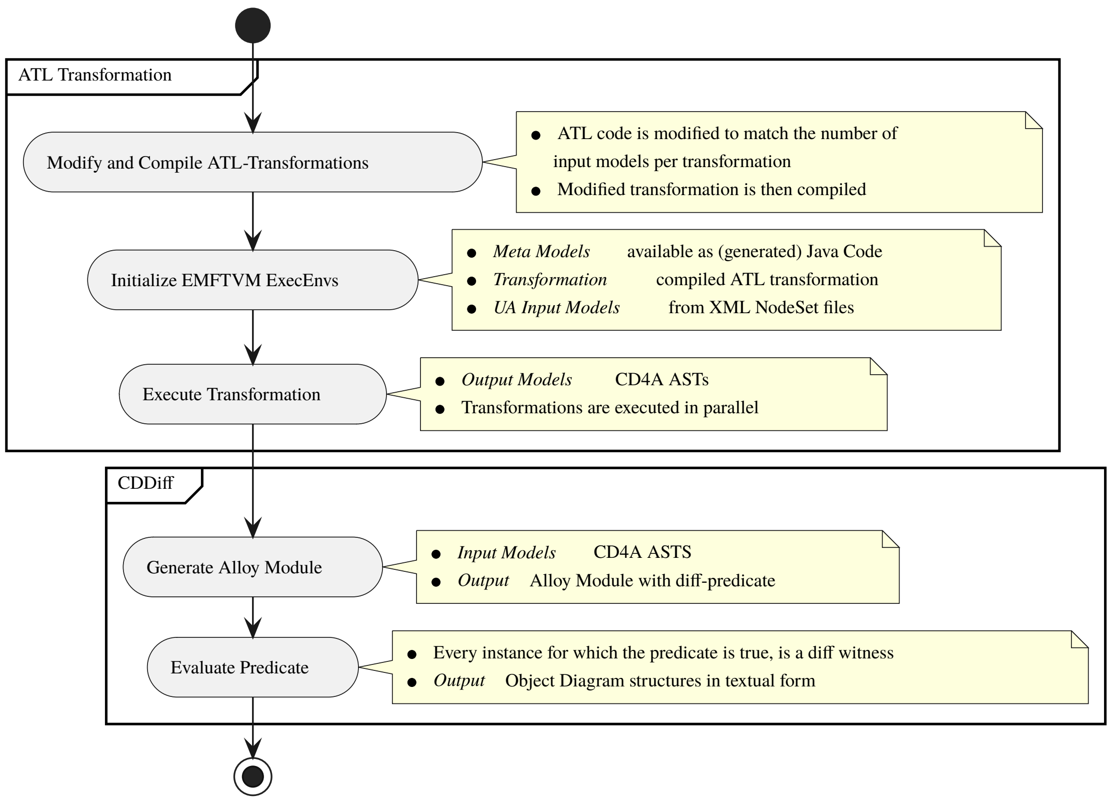

ATL Transformation
Modify and Compile ATL-Transformations
ATL code is modified to match the number of input models per transformation
Modified transformation is then compiled
Initialize EMFTVM ExecEnvs
Meta Models available as (generated) Java Code
Transformation compiled ATL transformation
UA Input Models from XML NodeSet files
Execute Transformation
Output Models CD4A ASTs
Transformations are executed in parallel
CDDiff
Generate Alloy Module
Input Models CD4A ASTS
Output Alloy Module with diff-predicate
Evaluate Predicate
Every instance for which the predicate is true, is a diff witness
Output Object Diagram structures in textual form

Figure B.2: Activity diagram showing the workflow of the *uadiff* application.

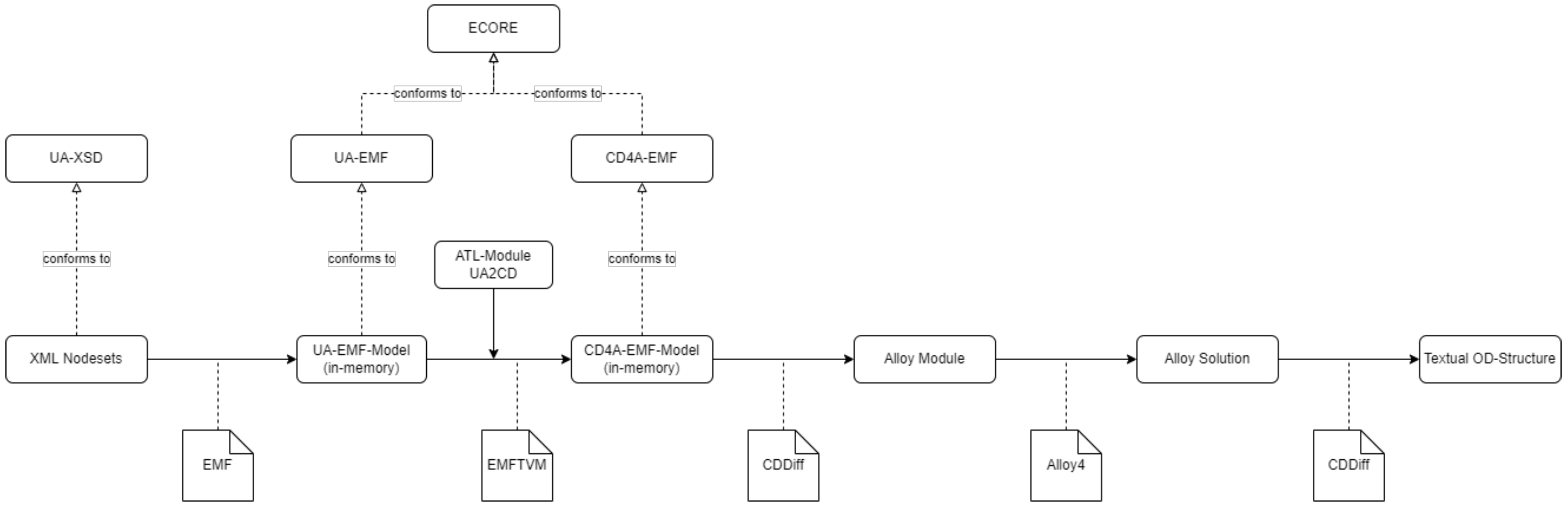


Figure B.3: Different model representations and transformations when running *uadiff*

C UA Graphical Notation

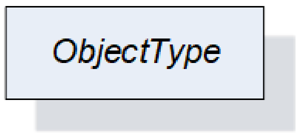
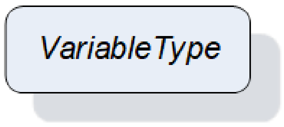
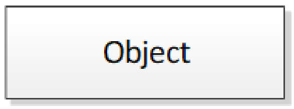
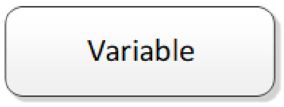
NodeClass	**Graphical Notation**
ObjectType	
VariableType	
Object	
Variable	

Table C.1: OPC UA graphical notation [OPC21b] for *NodeClasses* relevant to this work. Graphical elements from [OPC21a].

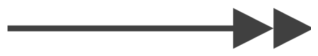
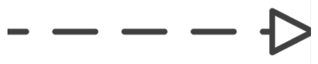
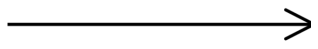
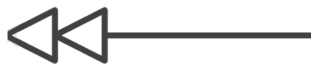
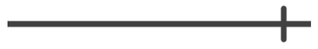
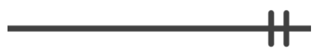
ReferenceType	**Graphical Notation**
HasTypeDefinition	
HasInterface	
HierarchicalReferences	
HasSubType	
HasComponent	
HasProperty	

Table C.2: OPC UA graphical notation [OPC21b] for *ReferenceTypes* relevant to this work. All *References* are forward *References* from left to right. Graphical elements from [OPC21a].

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature