Christian-Albrechts-University of Kiel
Department of Computer Science
Software Engineering Group

Automatic Conformance Checking of C#-based Software Systems for Cloud Migration

Master's Thesis

2012-03-29

Written by: Christian Wulf
born in Kiel on 1986-12-20

Supervised by: Prof. Dr. Wilhelm Hasselbring
M.Sc. Sören Frey

Hiermit versichere ich, Christian Wulf, dass ich die Masterarbeit selbständig verfasst und keine anderen als die angegebenen und bei Zitaten kenntlich gemachten Quellen und Hilfsmittel benutzt habe und die Arbeit in keinem anderen Prüfungsverfahren eingereicht habe.

Datum, Ort, Unterschrift

Abstract In recent years, cloud computing has emerged as a promising way to cost-efficiently use resources and to rapidly scale up and down according to the current workload. For this reason, many companies want to migrate their software systems to the cloud. However, all available cloud providers impose some restrictions on the systems that run within their cloud environments. Moreover, to efficiently utilize the cloud, the system's architecture often has to be changed.

To assist in migrating software systems to the cloud, Frey and Hasselbring developed the CloudMIG approach. It constitutes a programming language and cloud provider independent approach that utilizes the Knowledge Discovery Meta-Model (KDM). It describes in six steps how to systematically transform and adapt an existing system so that it is compatible with a given cloud provider and efficient according to user-defined rating criteria, e.g., performance or expenses. There also exists the application CloudMIG Xpress that implements the approach. Support for particular programming languages and cloud providers can be easily added by means of CloudMIG Xpress' plug-in architecture.

In this thesis, we add support for C#-based systems by providing a C#-to-KDM transformation. The development of the transformation thereby represents the main part of the thesis. However, we also offer a cloud profile for Microsoft Azure that describes the characteristics of the cloud provider, e.g., the pricing configuration, the supported programming languages, and the constraints already mentioned above. Furthermore, we implement an exemplary C# constraint validator that checks for write accesses to the local file system. Since Microsoft Azure discourages using the file system to store data, this validator can be used to validate the conformance with this cloud environment. Finally, we perform a performance and an accuracy analysis on the transformation and apply CloudMIG Xpress on an industrial example application the first time.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Cloud Computing has emerged as a new computing model where resources are provided as utilities. The potential benefits of scalability, elasticity, and resource efficiency are very attractive to many enterprises.

However, current software systems are often not designed to run in a cloud environment. The architecture of a system might not be laid out to be scalable or violates some constraints that the given cloud environment specification imposes. For example, storing data permanently in Microsoft Azure [2] requires the use of a Microsoft Azure drive because the local file system of a virtual machine (VM) does not represent a persistent storage resource. Even worse, it is often not known whether a software system is compatible with a given cloud environment at all.

In order to analyze and evaluate software systems to support cloud migration, Frey and Hasselbring [13] developed the CloudMIG approach. It defines six steps to semi-automatically transform a software system of any programming language to an arbitrary cloud environment. For this purpose, CloudMIG makes extensive use of the Knowledge Discovery Meta-Model (KDM) [17]. In the first step, it transforms the given system to an appropriate KDM instance, for example. Moreover, it provides an KDM-based Cloud Environment Model (CEM) that is able to describe arbitrary cloud environments, such as Amazon EC2 and Microsoft Azure.

Another step of the CloudMIG approach involves a conformance checking analysis [14, 15] of the given software system with respect to the selected cloud environment. Thereby, specific validators that check for cloud environment constraints are applied on the KDM representation of the system.

Frey and Hasselbring have already developed a prototype implementation called CloudMIG Xpress that performs the six steps that are mentioned above. Until now, it only supports Java-based software systems and two cloud profiles, one for Amazon EC2 and one for Google App Engine for Java. Hence, in order to demonstrate the generic applicability of this approach, we will integrate support for C#-based software systems and for Microsoft's cloud platform Azure. Since CloudMIG Xpress has been only applied on open-source software systems so far, we additionally carry out a first industrial case study.

1.2 Goals

The goal of the master's thesis is to integrate support for C# in CloudMIG Xpress and to evaluate CloudMIG's constraint violation detection approach. For this purpose, we develop a program that converts a C#-based software system into a corresponding KDM instance. Furthermore, we build a cloud profile that describes the constraints that the cloud provider Microsoft Azure defines.

Finally, we use this cloud profile and an example application, namely the financial risk assessment system *Nordic Analytics* (HSH Nordbank), to instruct CloudMIG to check the corresponding KDM instance for Microsoft Azure conformance.

1.2.1 Goal 1: C#-to-KDM Transformation

We first construct a Java-based transformation that parses C# source code by means of an appropriate ANTLR-conform C# grammar and transforms it into a KDM-conform model (see Section 3 and 4). This model will serve as the input for CloudMIG Xpress. Furthermore, our plug-in will also implement CloudMIG Xpress' and MoDisco's plug-in interfaces in order to directly integrate it into them (see Section 5).

1.2.2 Goal 2: Microsoft Azure Cloud Profile

In the next step, we build a cloud profile describing Microsoft's Azure cloud environment properties, especially its constraints (see Section 6). For this purpose, we use the cloud environment meta-model specification of CloudMIG Xpress.

1.2.3 Goal 3: C#-specific Constraint Validators

Consecutively, we construct further constraint validators that are specific to C# applications, e.g., constraints related to the read and write access to the operating system's file structure (see Section 7).

1.2.4 Goal 4: Evaluation of the Transformation and the Automatic Conformance Analysis

In the last step, we perform a performance and an accuracy analysis on the aforementioned transformation (see Section 8). Moreover, we evaluate CloudMIG Xpress'

conformance analysis by applying it on the financial risk assessment system *Nordic Analytics*. We analyze the detection of the constraint violations by means of the above steps.

1.3 Structure of the Thesis

The next section describes foundations and technologies that are necessary to understand and implement our goals. Section 3 presents the steps to build the ANTLR-conform grammar that is used to generate the C# parser and abstract syntax tree in Java. Subsequently, the transformation from C# to KDM is described in detail in Section 4. Section 5 introduces the plug-in interfaces for CloudMIG Xpress and MoDisco. In Section 6 and 7, the cloud profile for Microsoft Azure and a corresponding exemplary C#-specific constraint validator is described in detail. Section 8 evaluates the transformation described in Section 4 by performing a performance analysis and an accuracy analysis on it. Furthermore, it presents a conformance checking analysis of CloudMIG Xpress by means of *Nordic Analytics*. Afterwards, in Section 9, we describe related work. Section 10 concludes the thesis and gives an outlook on future work.

2 Foundations and Technologies

In the following, we describe the foundations and technologies that are used to achieve the goals defined in Section 1.2.

2.1 Foundations

Migrating a software system to the cloud requires knowledge of several areas. Thus, to get an understanding of them, we consider the most important ones in the next five subsections.

Reverse Engineering

Reverse engineering [6] is the process of analyzing a given system in order to identify its components and their associations, and to create appropriate representations or views of it. Such views often represent an abstraction of the system to focus on a particular aspect of the system.

Software architecture reconstruction (SAR) as part of the reverse engineering process can help in understanding and restructuring the system's architecture. It can reveal the architecture style and allow the reengineer to compare the conceptual and concrete architecture. [19] For such conformance checking analysis, the reengineer can utilize different architecture views and viewpoints. Furthermore, SAR additionally supports software evolution and maintenance, serves as foundations for reuse investigations, and can be used to control the architectural drift resulting from the co-evolution of the implementation and the conceptual architecture. [7]

In the context of this thesis, we reconstruct the architecture of C#-based software systems and utilize KDM to perform a conformance checking analysis according to the cloud environment Microsoft Azure.

Cloud Computing

Cloud Computing has evolved from the fact that most industrial data centers have a low resource utilization. They often use only up to 20% of available resources [1] while still consuming more than 50% of the energy necessary at peak times. Thus, such data centers have a high degree of over-provisioning beyond the periods of

peak workload. The more pronounced the variation of workload per time interval, the more the waste of resources.

Cloud Computing [32, 39] provides a promising concept to minimize such waste by offering the idle resources for further purposes. Depending on whether an organization offers their free resources to their subsidiaries and employees, to the general public, or to both, it serves as private, public, or hybrid cloud, respectively.

In the following, we will use the definition of Cloud Computing and related terms that Mell and Grance [22] propose and published at the National Institute of Standards and Technology (NIST) . Although there are many other definitions, the NIST definition becomes a de-facto standard.

Independent of the so-called cloud deployment model, the allocation of some resources of an Infrastructure-as-a-Service (IaaS) cloud is usually realized by virtualization. Thereby, a virtual machine (VM) represents an amount of resources, e.g. two 2.4 GHz processor cores with 2 GB RAM and 10 GB space capacity. In this way, a cloud user can consume such resources in isolation from other cloud users and without considering the underlying hardware.

This example demonstrates the usage of IaaS. However, there are several other so-called service models such as the Platform-as-a-Service (PaaS) and the Software-as-a-Service (SaaS) model. The former represents not only the basic infrastructure, but also a particular environment such as an Ubuntu operating system with an installed integrated development environment. Thus, the cloud user's scope of using the allocated resources is bound to such an environment. In the context of SaaS, the cloud user has only access to a particular software. Here, the resources are represented as the application or Application Programming Interface (API) that is provided by the so-called SaaS provider.

Resources are normally offered on a pay-for-use basis, i.e., the cloud user only pays for the actual amount of allocated resources and the demanded period of time. He himself controls the duration by starting and stopping his virtual machine or cloud application, respectively. Hence, Cloud Computing is also characterized by elasticity and flexibility. One can allocate resources on demand and release them when they are not needed anymore. This feature supports especially start-up companies that have initially not the (financial) resources to sustain high peak workloads.

Such companies do not need an up-front commitment, but can utilize the cloud's dynamic resource allocation.

The possibility to choose the type and amount of resources suggests cloud users of apparently infinite computing resources. Thus, they can utilize the cost associativity of a cloud to save time. For example, they can choose to allocate one server for 1,000 hours or 1,000 servers for one hour for the same price.

In conclusion, Cloud Computing does not try to only minimize resource wastage of data centers, but also to provide a way of cost-efficient resource allocation for cloud users.

Cloud Migration

Migrating an existing software system to the cloud requires several considerations and changes [1, 14]. First, an enterprise has to take organizational issues into account such as new or changing responsibilities for the billing and maintenance of the cloud instances. Second, sensible data are now no longer exclusively stored and processed on premise. Thus, the enterprise management has to consider liability and auditing issues. One way to handle these concerns it to identify such sensible externally processed data and apply some sort of cryptography on it. Third, there are several technical shortcomings of current cloud migration approaches. Most of them are restricted to a particular cloud provider and offer only a limited degree of automation. Moreover, only a few approaches focus on efficient resource allocation. Other approaches provide, if at all, only limited automation support for scalability evaluation at design time. Hence, a migration to the cloud requires an intensive planning and evaluation phase and can be quite time-consuming.

The CloudMIG Approach

The CloudMIG [13, 15] approach aims to overcome some limitations of current cloud migration approaches, namely the applicability, the level of automation, the inefficient resource allocation, and the lack of scalability. Today's solutions are most often limited to particular cloud providers. They are thus not able to migrate to an arbitrary cloud provider, but have to port to the given one. Furthermore, the target cloud-conform architecture cannot be (semi-)automatically generated so far. Additionally, the target architecture's violations against the cloud environment's

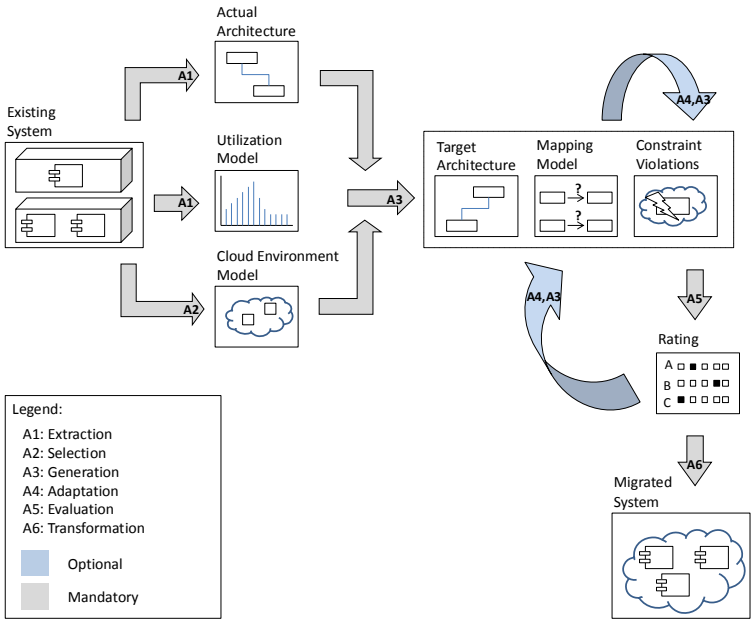


Figure 1: The CloudMIG Approach [13]

constraints are not recognized. Another obstacle is the resource inefficient design of most applications. They do not leverage the cloud environments' elasticity. Finally, no approach analyzes the target architecture's scalability at design time.

The CloudMIG approach defines six steps to migrate an arbitrary application into a user-defined cloud environment. We briefly describe them below. Figure 1 illustrates the approach.

A1 - Extraction The first step results in architectural and utilization models of the legacy system. For this purpose, the Knowledge Discovery Meta-Model (KDM) and the Structured Metrics Meta-Model (SMM) of the Object Management Group (OMG) are used as model specifications.

A2 - Selection The next step deals with selecting the target cloud provider by choosing an appropriate CEM-compatible cloud profile candidate.

A3 - Generation In this step, a constraint violation model is generated by means of the models from step 1 and the selected cloud profile from step 2. It describes the current system's violations against the constraints of the selected cloud environment.

It also serves as input for the generation of the target architecture. A mapping model maps the elements of the models from step 1 to the target architecture model.

A4 - Adaption This step is left for manual adaptations to the target architecture by a reengineer. Such non-automatic actions could be required if the removal of a violation is too complex or needs additional knowledge.

A5 - Evaluation The penultimate step involves an evaluation of the target architecture. For this purpose, static analysis and runtime simulation are applied. Furthermore, the target architecture is evaluated according to a rating based on specific, user-defined criteria.

A6 - Transformation The last step deals with the transformation from the generated and adapted target architecture to the selected cloud environment. This transformation, however, needs to be perform manually by the reengineer.

Parser Generation

Parser generation [11, 34] describes the generation of a parser from a grammar description. There are two popular types of parser generators, namely LL and LR parser generators. For the sake of completeness, we also refer to *Packrat* [12] and Generalized LR parsers [21].

LL parsers use parsing techniques that scan the input from left to right (the first L of LL) and produce a leftmost derivation (the second L of LL) of the initial grammar rule. A leftmost derivation strategy expands nonterminals (also called grammar rules) from left to right. LR parsers use parsing techniques that also scan input from left to right, but produce a rightmost derivation (the R of LR). A rightmost derivation strategy expands nonterminals from right to left.

Usually, LL parsers do not accept left-recursive rules and LR parsers do not accept right-recursive rules. Otherwise, the deviation strategy would cause an infinite rule invocation in both cases.

All generated parsers require some lookahead to decide which rule to choose and to prevent infinite processing of left- and right-recursive rules. For this reason, the type of a parser is appended by the maximal number of tokens that are used to

perform a lookahead. For example, an LL(2) parser additionally uses a maximum of 2 tokens after the current one.

The first popular parser generator was Yacc [16], an acronym for **Y**et **A**nother **C**ompiler-**C**ompiler. Yacc is an LR parser, originally written in portable C, that accepts some special LR(1) grammars.

2.2 Involved Technologies

We now consider the technologies that we use to implement the transformation, the cloud profile, and the constraint validators.

ANTLR

ANother Tool for Language Recognition (ANTLR) [27, 28, 29] is a framework for constructing parsers in target programming languages such as Java, C# , and Python from grammar descriptions. Furthermore, ANTLR provides support for tree construction, error recovery, and error reporting.

We use a C# grammar defined in ANTLR syntax in order to parse C# source code. We additionally adapt it to let ANTLR automatically generate a corresponding Abstract Syntax Tree (AST)in Java. In doing so, we can use Java to transform the C# program into a KDM instance. We do not want to mix the ANTLR-based grammar with Java syntax or XML/KDM notation and thus separate the processing in multiple steps.

Lexer and Parser Grammars ANTLR provides three different grammar types, namely lexer, parser, and tree grammars. The first is responsible for the lexical analysis. The second defines the syntactical analysis. The third can be used to check complex semantic constraints. However, tree grammars are rarely used because their's definition is more time-consuming. That is why we only describe the lexer and parser grammar in more detail.

The lexer grammar defines what sequences of characters correspond to what so-called tokens. Tokens are the input for the syntactical analysis, i.e., parser grammars base on lexer grammars. Listing 1 demonstrates a simple lexer grammar that recognizes the character sequences *true* and *false* as well as sequences of the numbers 0 to 9. Below, we discuss the example in more detail.

Listing 1: A Simple Lexer Grammar

```
1  lexer grammar Example;
2
3  options {
4    language = Java;
5  }
6
7  TRUE : 'true' ;
8  FALSE : 'false' ;
9  INTEGER_LITERAL :
     DECIMAL_DIGIT+ ;
10 fragment DECIMAL_DIGIT :
     '0' | '1' | '2' | '3' |
     '4' | '5' | '6' | '7' |
     '8' | '9' ;
```

Listing 2: Valid and Invalid Lexer Inputs

```
1 true          // valid
2 falsetrue9684   // valid
3 fa            // invalid
4 True          // invalid
5 1234 true       // invalid
```

The first non-comment-statement of a grammar defines the type (lexer, parser, or tree[1]) and the name of it. The grammar's name must correspond to it's file name. In our example, we define a lexer grammar with the name `Example`. In the `options` section, we can specify options for the output generation. When executing ANTLR on our lexer grammar, it will generate a complete so-called Lexer in the target programming language Java because the options' `language` attribute is set to `Java`.

Our lexer contains four rules (also called lexer rules). Lexer rules always start with a capitalized letter. The first one, `TRUE`, in line 7 defines a new token of type TRUE that is created each time the character sequence "true" is recognized by the lexer. The second one, `FALSE`, in line 8 defines a new token of type FALSE that is created each time the character sequence "false" is recognized. In line 9, the token `INTEGER_LITERAL` is defined by means of another lexer rule called `DECIMAL_DIGIT`. The + thereby denotes the positive closure, i.e., it matches the previous token one or more times. There is also the Kleene closure operator * that matches zero or more token occurrences. In our example, `INTEGER_LITERAL` is created each time `DECIMAL_DIGIT` can be applied one or more times. The token `DECIMAL_DIGIT` is created if the character '0', '1', '2', .., or '9' is detected. The keyword `fragment` indicates that when matching `DECIMAL_DIGIT` no corresponding token should be

[1]Tree grammars base on parser grammars. Their inputs are parser rules. However, this type of grammar is rarely used and not relevant for our purpose.

Listing 3: A Parser Rule for Namespace Declarations

```
1  namespace_declaration
2    : NAMESPACE qualified_identifier namespace_body SEMICOLON
       ?
3    ;
```

created. It is used for performance optimization reasons. In the following, we sometimes call these lexer rules just *fragments*.

Listing 2 shows five different character sequences. The first and second ones are valid with respect to our lexer grammar. `true` would be matched by the lexer rule `TRUE` and transformed into a TRUE token. `falsetrue9684` would be matched by the lexer rules `FALSE`, `TRUE`, and `INTEGER_LITERAL` in that order and transformed into the three corresponding tokens of the same name. The remaining three character sequences, however, are not valid. There is no lexer rule that matches `fa` because rules match completely or not at all. `True` cannot be matched either since rule definitions are case-sensitive. Although `1234` is matched by `INTEGER_LITERAL`, there is no rule for the following space character.

The parser grammar defines valid orders of tokens. The aim of the corresponding parser is to verify the correctness of the given token sequence with respect to the parser grammar. For this purpose, parser rules are used. Parser rules always start with uncapitalized letters in contrast to lexer rules. Listing 3 illustrates a parser rule that is responsible for matching C# namespaces.

The rule expects three to four symbols, each of them is either a terminal or nonterminal symbol. As indicated by the capitalized initial letter, `NAMESPACE` is a token, and therefore a terminal produced by the previous lexical analysis. `qualified_identifier` and `namespace_body` represent parser rules, i.e., nonterminal symbols.[2] After correctly parsing the previous three symbols, the whole parser rule `namespace_declaration` has already matched. The last terminal `SEMICOLON` is optional as indicated by the question mark.

If both grammars are valid, ANTLR is able to automatically generate a lexer and a parser from them in the user-defined target programming language.

[2]In this example, the definitions of the nonterminals are omitted for the sake of simplicity.

Listing 4: The ANTLR operators ^ and !

```
1 enum_definition
2   : ENUM^ IDENTIFIER enum_base? enum_body SEMICOLON!?
3   ;
```

Listing 5: An Example Rule Using a Rewrite Rule

```
1 property_declaration
2   : member_name OPEN_BRACE accessor_declarations CLOSE_BRACE
3     -> ^(PROPERTY_DECL  member_name  accessor_declarations)
4   ;
```

AST Generation Let us consider the AST generation in more detail. ANTLR offers a feature that automatically creates an AST while parsing when the `output` attribute in the `options` section of the parser grammar is set to `AST`. For each token the parser reads in, a new tree node is created. However, all nodes are arranged in sequence, i.e., no node hierarchy is automatically produced. Consequently, the tree consists of only two levels, one for the root node and one for all the token nodes. Thus, we have to define subtrees and remove some sort of tokens that are irrelevant for our purposes.

ANTLR provides two mechanisms to specify hierarchy and to remove nodes. First, one can use the operators ^ and ! on terminals and nonterminals. By using ^ after a token, ANTLR makes it the root node within the current subtree. If one uses the root operator on a rule, the first token within this rule is set to the root.

The ! operator is responsible for removing tokens. When defined after a token, that token is not created for the AST. If the operator is defined after a rule, all tokens of the rule are not considered when building the AST.

Listing 4 shows an example application of the operators. The `ENUM` token is set as the root containing all following tokens as children. As indicated by the ! operator, a corresponding `SEMICOLON` node is not created if it was parsed.

The other mechanism to define hierarchy and to suppress nodes in an AST is implemented by so-called *Rewrite Rules*. A rewrite rule rewrites a sequence of parsed terminals and nonterminals and creates the corresponding nodes within the current subtree of the AST. Listing 5 illustrates an example.

The right-arrow causes the generated parser to build the AST on the basis of the tokens and rules from the right-hand side instead of the left-hand side of the arrow. The ^ symbol is again used to define hierarchy in a slightly different way. In order to avoid adding a node for a particular token, one only needs to omit writing the token on the right-hand side. In our example, the generated parser would not build a corresponding node for the tokens `OPEN_BRACE` and `CLOSE_BRACE`. Furthermore, rewrite rules allow to add additional imaginary tokens. An imaginary token is a token that has no character representation. It is used to further group a set of nodes in an AST. Here, `PROPERTY_DECL` is one example of an imaginary token.

Ecore

Ecore is an object-oriented meta-model specification developed as part of the Eclipse Modeling Framework [9]. Eclipse Modeling Framework (EMF) provides sophisticated tool support to create an Ecore-based model, parse a file containing an Ecore-based model, and visualize an Ecore-based model.

There is also a KDM specification[3] defined as an Ecore-based meta-model. We use it in the extraction process to build the KDM representation of the C# example application.

Moreover, CloudMIG's cloud environment meta-model is also based on the Ecore specification. We use it to build a cloud profile that reflects the constraints of Microsoft's Azure cloud environment.

Knowledge Discovery Meta-Model (KDM)

The KDM [26] is a specification for representing information related to existing software systems. The OMG released the first version in 2008 and thereby provided a common interchange format, i.e., a language-independent representation of the source code, for instance.

KDM is structured in a hierarchy of four layers. Figure 2 illustrates these layers, their corresponding packages, and their concerns.

The lowest layer, called *Infrastructure Layer*, consists of the three packages *Core*, *KDM*, and *Source*. It represents common meta-model elements for higher

[3] `http://www.omg.org/spec/KDM/20090502/09-05-23.ecore`, last access on 2011-09-13

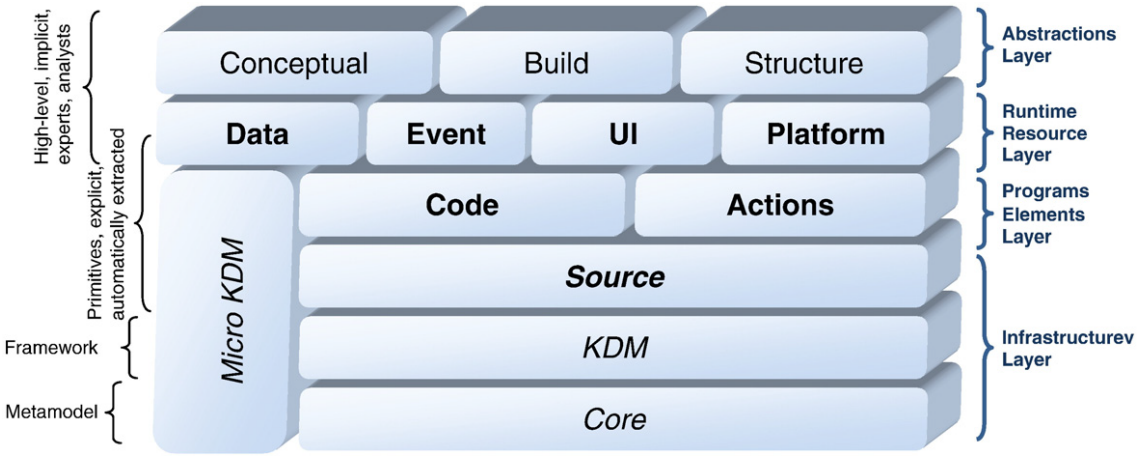


Figure 2: Layers, packages, and concerns in the KDM [31]

layers and describes the physical artifacts of the software system, e.g., source code and build files. This layer depends on none of the other layers.

The next overlying layer is called the *Program Elements Layer*. It represents the abstraction from the original language syntax of the given software system to the KDM language-independent format.

The *Runtime Resource Layer* defines patterns for representing the operating environment in which the given software system runs. For this purpose, it can also contain high-level knowledge, e.g., for some particular views that require some manual expertise.

The *Abstractions Layer* represents domain-specific and application-specific abstractions as well as artifacts concerning the build process. This layer defines the three KDM packages *Structure*, *Conceptual*, and *Build*.

We will focus on the *Infrastructure Layer* and the *Program Elements Layer*, especially on the source package, code package, and action package.

Source Package The source package provides model elements to represent the physical structure of the software system. It offers a representation for directories and several kind of files, for example source files, binary files, or image files. Additionally, it includes elements to link an element of a higher layer to a file or line in a file.

Code Package The code package comprises elements that represent the source code of the given software system independent of the used programming languages. For instance, it provides representations of classes, methods, variables, inheritance relationships, and modules.

Action Package The action package offers model elements that represent the parts of the source code that are responsible for the behavior of the software system. For this purpose, there are elements for operations, assignments, method calls, read/write accesses, the control flow, and conditions, for example.

We utilize KDM within the CloudMIG approach. CloudMIG expects a KDM-conform input model for the architecture of the considered software system to perform several analyses and architecture reconfigurations on it.

MoDisco

Many development tools are often specialized on generating one special model format (e.g., UML) from a given input (e.g. source code) and vice versa. They do not consider other model formats (e.g., XML, SQL, and KDM).

MoDisco [5], a Java framework for model-driven reverse engineering, aims at overcoming this homogeneity by providing a generic and extensible architecture. It offers flexible interfaces for all kinds of possible input legacy artifacts as well as different types of outputs depending on the given reverse engineering objectives. Furthermore, it allows to easily define a transformation from one or more input models to one or more output artifacts. Bruneliere et al. [5] call such a transformation a discoverer.

Although MoDisco includes some input and output model implementations as well as some discoverers today, it lacks in a discoverer that reads C# source code and transforms it to a corresponding KDM instance. The implementation of such a KDM discoverer is part of the master thesis.

Microsoft Azure

The Azure platform [2] is Microsoft's cloud environment and follows the PaaS model. It provides hardware, software, network, as well as storage resources and consists

of the three main components Windows Azure, SQL Azure, and the AppFabric. The individual components are described in the next paragraphs. Developers can build and deploy their applications on the Azure platform to utilize services such as automatic resource and life cycle management as well as load balancing.

Windows Azure supports general-purpose windows applications, especially such ones that are built on the .NET framework (e.g., using C# and managed C++) or other languages such as PHP, Ruby, Python, or Java. Each application needs to be assigned to one or more web role, worker role, and/or VM role instances.[4] The former role is customized for web application programming, e.g., it can accept HTTP(S) requests and has already installed Microsoft's Internet Information Services (IIS). The worker role is not exclusively designed for web application programming. Instead it can be used for common application development and may perform background processing for a web role, for instance. The VM role represents a VM image that can be deployed on Windows Azure. This role provides the most flexibility in configuring and controlling the execution environment. While the web and worker role run within a virtual machine, the VM role defines the virtual machine.

SQL Azure represents Microsoft's database management system for the cloud and is accessible via .NET and other windows interfaces. The AppFabric is responsible for easily creating distributed applications.

All applications, virtual machines, and physical resources are monitored and scaled by the so-called fabric controller. For this purpose, the user needs to upload an XML-configuration file that describes what components and services the application provides. By means of this file, the fabric controller decides on which servers the application should run to provide optimal hardware utilization and fault tolerance.

.NET Framework

The .NET framework [25] constitutes Microsoft's implementation of the Common Language Infrastructure (CLI) specification and represents a programming model that provides standard solutions to facilitate the development of Windows applications. It consists of a runtime environment, the so-called Common Language Runtime (CLR), with a Just-In-Time (JIT) compiler, a collection of libraries, common interfaces, and service applications. For example, the framework contains a class

[4] `http://msdn.microsoft.com/en-us/library/windowsazure/gg432976.aspx` (2012-03-28)

Listing 6: An *HelloWorld* Example in C#

```
1  namespace ExampleNamespace
2  {
3      class HelloWorld
4      {
5          static void Main(string[] args)
6          {
7              System.Console.WriteLine("Hello␣World!");
8          }
9      }
10 }
```

`System.IO.File` that provides methods for the creation, copying, deletion, moving, and opening of files.

.NET-based applications are compiled to the Common Intermediate Language (CIL), a hardware-independent intermediate code. In order to execution code in the CIL format, the CLR needs to be installed on the target system. It handles the loading, interpretation, and execution of CIL-based code. While executed, the CIL code is finally compiled by the JIT compiler to the actual processor dialect.

Listing 6 shows an *HelloWorld* example in C# that makes use of the .NET class `System.Console`.

As Windows Azure extensively uses the .NET framework for the communication and the access on I/O devices, it can pose some .NET-specific constraints. For such constraints, we develop specific validators (see Section 7).

CloudMIG Xpress

There also exists a prototype implementation of the CloudMIG approach, called CloudMIG Xpress. It supports the steps 1-5 and provides a flexible architecture to add new cloud profiles, for example. The last step is not supported and therefore needs manual operations.

3 Building an Appropriate C# Grammar

In order to define a transformation that maps a C# application to a corresponding KDM-conform representation, we must first be able to parse C# source files. For this purpose, we need a Java-based C# parser because we finally want to integrate the transformation into CloudMIG Xpress (see Section 1.2) that is implemented in Java.

We choose the very popular and matured parser generator ANTLR that is able to generate a parser in several programming languages including Java.[5] We only need a C# grammar file in an ANTLR-specific syntax. One candidate is the BNF-based grammar defined in the C# specification. Although the ANTLR syntax is similar to the Extended Backus–Naur Form (EBNF) [33] and hence also to BNF, it is not the same. Furthermore, the C# grammar contains some informal rule definitions such as phrases like "one of the following characters [...]" and "characters from unicode class Z". But even though it would conform to the syntax, one has further to consider the fact that ANTLR is an LL(*) parser and thus does not accept left-recursive rule definitions (see Section 2.1). That is why we cannot directly use it as input for ANTLR.

Since C# has a complex semantics, we want to use a mature ANTLR-based C# grammar instead of writing one completely by our own. Thus, we evaluate available C# grammars in Section 3.2. But first, we want to motivate the need of a C# preprocessor in addition to a grammar in Section 3.1. In Section 3.3, we then discuss and describe further necessary adaptations and extensions to the grammar. Subsequently, we define and build an ASTon the basis of the grammar in Section 3.4.

3.1 Necessity of a C# Preprocessor

Preprocessor directives are an integral part of C#. They enable conditional compilation, let the programmer define convenient collapsible regions of source code, and allow to specify custom compiler messages. Due to the first aspect, we require a preprocessor in addition to the grammar. The following examples motivate this necessity:

[5] We discuss alternative parser generators in Section 9.

Listing 7: Simple example

```
1 #if COND_A
2   a = 0;
3 #elsif
4   a = 1;
5 #endif
```

Listing 8: Advanced example

```
1 if (cond) {
2   b = 0;
3 #if COND_B
4 } else {
5   b = 1;
6 }
7 #elsif
8 }
9 #endif
```

Both listings illustrate the use of conditional preprocessor directives. Listing 7 shows some C# source code that is compiled either to `a=0;` or `a=1;` depending on the macro variable `COND_A`. In particular, it is not allowed to compile both statements. A lexer that completely conforms to the C# specification would first evaluate the condition of the if-directive and then generate the corresponding tokens. For example, if `COND_A` is `true`, the lexer would generate the tokens `a`, `=`, `0`, and `;`. If a lexer just skips all preprocessor directives and does not evaluate if-directives, it would generate too many tokens concerning the C# specification. In the worst case, this can lead to syntax errors (see discussion of Listing 8). In Listing 7, the parser would get the tokens from line 2 and 4 from the lexer and detect two statements. Hence, the snippet of source code would not be processed correctly, even though the resulting tokens conform to the C# syntax.

Listing 8 shows an advanced example of conditional preprocessor directives. Here, the body of the if-statement is closed depending on `COND_B`. According to the C# specification, the source code snippet is interpreted as followed: Either the body is closed and an else-statement is appended or it is simply closed without appending an else-statement. If a non-conform lexer again just skips the preprocessor directives without evaluating, the parser would complain about a syntax error. Both code sections enclosed by the conditional directives contain the closing token `}` for the if-statement (line 4 and 8).

Thus, in order to parse an arbitrary C# software system, we must also incorporate a preprocessor. Since a preprocessor must evaluate conditions while scanning and producing tokens, we need some kind of logic in addition to the pure grammar

Listing 9: Example of an Action Code Section

```
floating_point_type
  : FLOAT
    {System.out.println("parsed␣float␣token");}
  | DOUBLE
    {System.out.println("parsed␣double␣token");}
  ;
```

Listing 10: A lexer's if-directive rule

```
[...]
Pp_if_section
@init {Expression exprObj = new Expression();}
  : WHITESPACE? SHARP WHITESPACE? 'if' WHITESPACE
     Pp_expression[exprObj] Pp_new_line
    {push(exprObj.isTrue());}
  ;
Pp_endif
  : WHITESPACE? SHARP WHITESPACE? 'endif' Pp_new_line
    {pop();}
  ;
[...]
```

definition syntax of ANTLR. For this purpose, ANTLR provides the concept of the so-called *action code sections*. Action code sections allow to embed arbitrary source code in the target language into a grammar definition file. They can be inserted at arbitrary positions, e.g., outside and inside grammar rules.

Listing 9 shows a parser rule that matches either exactly one `FLOAT` token or exactly one `DOUBLE` token. Additionally, *"parsed float token"* or *"parsed double token"* is displayed, respectively, due to the embedded method invocation of `System.out.println`.

By using ANTLR's action code mechanism, we can insert the necessary logic for a preprocessor to the preprocessor directive rules. Listing 10 shows a lexer rule that is responsible for matching the if-directive and creating a corresponding token named after the rule name, i.e., `Pp_if_section`. When the lexer checks whether the `Pp_if_section` rule matches the next input sequence, it first processes the @init action code section. Here, an instance of the class `Expression` is created. Second,

the lexer successively checks each subrule whether it matches or not. If not, it aborts and checks the next lexer rule in the grammar file (in this case `Pp_endif`). Otherwise, when reaching the `Pp_expression` subrule, the lexer passes the `Expression` instance, created above, to the subrule before invoking it. `Pp_expression` in turn executes its subrules that are responsible to match and evaluate expressions like *or*, *and*, *equal*, *unequal*, *not*, *true*, *false*, *boolean macro variables*, and *parentheses expressions*. The result is passed and saved in the given `Expression` instance. If the `Pp_if_section` successfully invoked its subrule `Pp_new_line`, the subsequent action code in line 5 is executed. Here, the `Expression` instance's method `isTrue` returns the evaluated expression as boolean value. The `push` method puts this value onto a stack that realizes the evaluation of nested conditional directives. If later the corresponding endif-directive is matched by the `Pp_endif` rule, the value is removed from the stack by invoking the method `pop` within the action code section.

3.2 Comparison of Available C# Grammars

We searched the internet for free C# grammars in the ANTLR format by using Google with the keywords *antlr*, *C#*, and *grammar*. Table 1 lists all grammars of the first 100 results. The table's order corresponds to the order of the results.

Supported C# Version	**Required ANTLR Version**	**Contains Preprocessor Implementation?**	**Author**	**URL**	**Last Update**
4	3.2-3.4	yes	Andrew Bradnan	[a]	2010-06-20
1	2.0	no	Quentin Gregory	[b]	2006-06-29
2	2.0	no	Todd King	[c]	2005-09-26
4	3.0-3.4	no	Lucian Wischik	[d]	2010-04-19

Table 1: Available C# Grammars

[a] `http://antlrcsharp.codeplex.com/` (2012-03-10)
[b] `http://www.antlr.org/grammar/1151612545460/CSharpParser.g` (2012-03-10)
[c] `http://www.antlr.org/grammar/1127720913326/tkCSharp.g` (2012-03-10)
[d] `http://blogs.msdn.com/b/lucian/archive/2010/04/19/grammar.aspx` (2012-03-10)

The first column describes what C# version is supported by the given grammar. The second column shows the required ANTLR version to compile the gram-

mar. Column number three describes whether the author also provides an implementation for the preprocessor. We discuss this point in the next paragraph in more detail. The fourth and fifth column show the author and the source of the grammar, respectively. The last column contains the date of the last update of the grammar.

Since the current versions of *Nordic Analytics* and *SharpDevelop* require C# version 4, we cannot use the second and third grammar to parse and to build an AST. Moreover, a simple evaluation shows that these two grammars contain bugs and are restricted to particular use cases, for example, they do not include generics.[6] Additionally, one grammar already contains some AST nodes, most of them are not necessary for our purpose or contain insufficient information, i.e., they are too abstract. Thus, we consider the two grammars that are left.

Analysis of the C# grammar by Andrew Bradnan

The grammar written by Andrew Bradnan looks promising since it not only implements the full C# specification version 4, but also comes with a preprocessor implementation. Furthermore the author delivers unit tests for his grammar including his own small unit test framework.

Bradnan's grammar is a combined grammar, i.e., it contains both the lexer grammar and the parser grammar in one file. The target language is set to C# since all action code sections consist of C# source code.

If we want to use this grammar, we would have to at least set the target language to Java and translate the action code sections to Java equivalent statements. The author uses action code for the preprocessor implementation in the lexer section, for instance. However, he does not fully implement the preprocessor logic. For example, Listing 11 shows that in the action code of the lexer rule `PP_UNARY_EXPRESSION` the current value on the stack is negated and then again pushed onto the stack. This correctly implements a boolean *not*. Though, the rules `PP_OR_EXPRESSION` and `PP_AND_EXPRESSION` do not contain any action code sections that are responsible for boolean *or* and *and* processing, respectively.

Furthermore, the author changed the structure of the original C# grammar specification not only to make it ANTLR-conform, but also for performance optimization reasons. For this purpose, he splits some parser rules up, removes back-

[6] Generics are introduced in C# version 2.

Listing 11: Incomplete Preprocessor Implementation

```
1  PP_OR_EXPRESSION
2    : PP_AND_EXPRESSION   TS*   ('||'   TS*   PP_AND_EXPRESSION
         TS* )*
3    ;
4  PP_AND_EXPRESSION
5    : PP_EQUALITY_EXPRESSION   TS*   ('&&'   TS*
       PP_EQUALITY_EXPRESSION   TS* )*
6    ;
7  PP_UNARY_EXPRESSION
8    : PP_PRIMARY_EXPRESSION
9    | '!'   TS*  PP_UNARY_EXPRESSION  { Returns.Push(!Returns.
       Pop()); }
10   ;
```

tracking by using syntactic and semantic predicates instead, and generalizes some other parser rules.

Listing 12 illustrates the most generalized and therefore most important rule `primary_expression` for expression recognition in Bradnan's grammar. Listing 13 shows the original version of this rule. Both versions are slightly simplified in syntax and code size so that the reader can follow more easily.

If we compare both versions with each other, we can see that the single rule `creation_expression` in line 2 of Listing 12 replaces the three original rules `object_creation_expression`, `delegate_creation_expression`, and `anonymous_object_creation_expression` in line 12-14 of Listing 13. Even more invasive is the introduction of the two rules `primary_expression_start` and `primary_expression_part`. They contain the semantics of the original expressions from line 1 to 9. In particular, the four original rules from line 4 to 7 are not explicitly declared anymore. For example, a member access like `System.Console.BufferHeight` as well as a method invocation like `System.Console.OpenStandardError()` is covered by the same sequence of rules, namely one time `primary_expression_start` for the first identifier and two times `primary_expression_part` for the dot followed by another identifier.

All these adaptations make it very difficult to read and navigate in the grammar and to later define AST nodes for the desired grammar rules.

Listing 12: Bradnan's `primary_expression` (simplified)

```
1  primary_expression
2    : creation_expression
3    | primary_expression_start
         primary_expression_part*
4    | sizeof_expression
5    | checked_expression
6    | unchecked_expression
7    | default_value_expression
8    | anonymous_method_expression
9    ;
10 primary_expression_start
11   : predefined_type
12   | (identifier     '<') => identifier
           generic_argument_list
13   | identifier ('::'    identifier)?
14   | 'this'
15   | 'base'
16   | paren_expression
17   | typeof_expression
18   | literal
19   ;
20 primary_expression_part
21   : access_identifier    // represents
         '.' identifier
22   | brackets_or_arguments
23   ;
```

Listing 13: Original `primary_expression` (simplified)

```
1  primary_expression
2    : literal
3    | parenthesized_expression
4    | typeof_expression
5    | simple_name
6    | member_access
7    | invocation_expression
8    | element_access
9    | this_access
10   | base_access
11   | post_increment_expression
12   | post_decrement_expression
13   | object_creation_expression
14   | delegate_creation_expression
15   | anonymous_object_creation_expression
16   | checked_expression
17   | unchecked_expression
18   | default_value_expression
19   | anonymous_method_expression
20   ;
```

Listing 14: True Negative Parsing Examples

```
1 if (value < 1) {}
2 if (value < 1 + 2) {}
```

Even worse, we found some valid C# source code that causes syntax errors when using the lexer and parser generated from Bradnan's grammar. Listing 14 shows two exemplary C# statements that are valid concerning the C# specification but result in parsing errors when using his grammar.

The if-statements in line 1 and 2 cause the error message "no viable alternative at input '1'". This means that the parser cannot find any rules within the current context that is able to match the input '1'.

At this point, we look more closely at the test framework that Bradnan delivers. We add new tests consisting of the statements from above and execute them by means of Bradnan's test framework. All tests wrongly pass without any error message. Thus, we can not rely on the results anymore and from now on doubt

the correctness and completeness of the whole grammar. Perhaps, it would have been better if the author had used the unit test framework gUnit for ANTLR-based grammars instead of writing one by himself.[7]

Analysis of the C# grammar by Lucian Wischik

Since all the grammars above do not satisfy our requirements of a correct and complete ANTLR-based implementation of the C# specification, we consider the grammar of Lucian Wischik, an employee at Microsoft. It is an almost direct translation of the grammar notation used in the C# and VisualBasic specification. The author specifically builds a program that transforms notations like $rule_{opt}$ and *an-example-rule* from the specification to the corresponding ANTLR-conform notations *rule*? and *an_example_rule*, respectively. To that extent, we already have an ANTLR-based C# grammar that is correct and complete according to the specification.

However, the resulting grammar file as a whole does not yet conform to the ANTLR format for the reasons already described at the beginning of this section. There are still some informal phrases and left-recursive rules. Hence, we make some adaptations to it so that it can serve as a valid input for ANTLR.

3.3 Grammar Adaptation

Since none of the ANTLR-conform grammars from above is a correct, complete, and C# 4-compliant one, we rather choose the correct, complete, and C# 4-compliant grammar from Wischik that, however, is not yet conform to ANTLR. Thus, we have to make several changes and adaptations.

Dividing the Original Grammar

We start by dividing the single grammar file into two parts: One for the lexical analysis and one for the syntactical analysis. In ANTLR, the first part is called the *lexer grammar*, the second part is called the *parser grammar*.

The authors of the grammar defined in the C# specification do not consider that the lexer and parser grammars are different and independent of each other

[7] `http://www.antlr.org/wiki/display/ANTLR3/gUnit+-+Grammar+Unit+Testing` (2012-03-10)

Listing 15: Informal Phrases in Rule `Whitespace_character`

```
1 Whitespace_character
2   : '<Any Character With Unicode Class Zs>'
3   | '<Horizontal Tab Character (U+0009)>'
4   | '<Vertical Tab Character (U+000B)>'
5   | '<Form Feed Character (U+000C)>'
6   ;
```

Listing 16: Resolved Informal Phrases

```
1 Whitespace_character
2   : UNICODE_CLASS_Zs //'<Any Character With Unicode Class Zs>'
3   | '\u0009' //'<Horizontal Tab Character (U+0009)>'
4   | '\u000B' //'<Vertical Tab Character (U+000B)>'
5   | '\u000C' //'<Form Feed Character (U+000C)>'
6   ;
```

(except for the fact that a parser grammar depends on a lexer grammar). For this reason, the lexer grammar, as translated by Wischik's program, contains some rules that actually belong to the parser grammar (c.f. the diverse `input*` rules in the C# specification). Furthermore, all rules are uncapitalized.

Changes to the Lexer

Hence, we move and remove some lexer rules. Moreover, we write a small program that capitalizes each rule in the lexer grammar. We then manually prepend the keyword `fragment` to those lexer rules that exist only for the sake of modularization and readability. Additionally, we add an individual lexer rule for each C# keyword, operator, and punctuator. In doing so, the parser can distinguish them by means of their different token types. For resolving the informal phrases, we replace each one with the corresponding tokens or fragments, respectively.

Listing 15 illustrates a lexer rule that is responsible for whitespace characters. It represents the original version as it is defined in the C# specification. The rule is specified only by informal phrases. Listing 16 shows our adapted version. In contrast, we specify the corresponding characters in the formal ANTLR syntax.

Listing 17: The Unicode Character Class Z in ANTLR syntax (excerpt)

```
fragment UNICODE_CLASS_Zs
  : '\u0020' // SPACE
  | '\u00A0' // NO_BREAK SPACE
  | [...]
  | '\u3000' // IDEOGRAPHIC SPACE
  | '\u205F' // MEDIUM MATHEMATICAL SPACE
  ;
```

Listing 18: A Left-Recursive Rule

```
namespace_or_type_name
  : IDENTIFIER type_argument_list?
  | namespace_or_type_name '.' IDENTIFIER type_argument_list?
  | qualified_alias_member
  ;
```

We left the previous informal phrases as comments at the right side. The last three characters are almost direct translations of the previous phrases. For the unicode character class Z, we add a fragment rule with the whitespace characters defined in the unicode standard [36] (see Listing 17).

We further have to resolve the left-recursive rules. Although ANTLRWorks[4], a special Integrated Development Environment (IDE) for ANTLR grammars, should be able to remove simple left-recursion, it does not work with our grammar. In addition, many left-recursive rules are not simple, e.g., the recursion takes place in the fourth or even deeper level of subrule invocation, especially in the parser grammar. Thus, we wrote another program that transforms simple and more complicated left-recursive rules into non-recursive rules by means of left-factoring. Listing 18 shows an example of a non-trivial left-recursive rule that we discuss in more detail below.

The parser rule `namespace_or_type_name` has three possibilities to be matched. The second one starts with a recursive call. In order to get a correct non-recursive version, we first have to append the rest of the second possibility, namely `'.' IDENTIFIER type_argument_list?`, to all other ones. Now we can remove the second one. Subsequently, we only have to enclose the rests with parentheses and add the Kleene closure operator to each of them. A slightly compacter non-recursive version

Listing 19: The Transformed Non-Recursive Rule

```
1 namespace_or_type_name
2   : (IDENTIFIER type_argument_list?
3   | qualified_alias_member
4   ) ('.' IDENTIFIER type_argument_list?)*
5   ;
```

Listing 20: An Unoptimized Rule

```
1 fragment Input_characters
2   : Input_character
3   | Input_characters
      Input_character
4   ;
5 // non-recursive version
6 fragment Input_characters
7   : Input_character (
      Input_character)*
8   ;
```

Listing 21: The Optimized Version of the Rule

```
1 fragment Input_characters
2   : Input_character+
3   ;
```

is demonstrated in Listing 19. Thereby, code duplication is avoided by means of further parentheses around the two remaining possibilities.

For increasing readability, we also build a program that optimizes both the lexer and the parser grammar. Besides having all capabilities of the EBNF standard, ANTLR grammars can additionally express some rules in a more compact and therefore more readable way. Listing 20 shows the left-recursive lexer fragment `Input_characters` and below a corresponding non-recursive version of it.

Listing 21 illustrates a further optimized version that utilizes ANTLR's positive closure operator. Since the subrule `Input_character` must occur at least one time, we are allowed to use `+` to replace the rule definitions from Listing 20 by the semantically equivalent version from Listing 21.

Implementing the Preprocessor

Since our chosen grammar does not contain a preprocessor implementation, we have to write one by ourselves. For this purpose, we also have to use some piece of Java code to implement the corresponding logic already mentioned above.

Listing 22: A Rule matching Almost Any Input

```
1 Input_character
2   : ~NEW_LINE_CHARACTER
      //'<Any Unicode
      Character Except A
      NEW_LINE_CHARACTER>'
3   ;
```

Listing 23: The Preprocessor Root Rule

```
1 Pp_directive
2   : (Pp_declaration
3   | Pp_conditional
4   | Pp_line
5   | Pp_diagnostic
6   | Pp_region
7   | Pp_pragma
8   ) {$channel=HIDDEN; }
9   ;
```

First, we have to sort our lexer rules. For a given input, ANTLR tries to match it using the rules of the grammar in the order in which the rules are specified. For example, if a rule such as `Input_character` illustrated in Listing 22 would be the first rule defined in our grammar, the lexer would almost only produce `Input_character` tokens because it matches any unicode character except a newline character.

In such situations, ANTLR complains with a similar warning message like "The following token definitions can never be matched because prior tokens match the same input."

Since, preprocessor directives have the highest priority concerning to the C# specification, we have to put the preprocessor directive root rule `Pp_directive` at the top of the grammar. After ordering some other rules until ANTLR does not complain anymore, we can begin implementing the preprocessor in a second step.

We start by inserting the action code `$channel=HIDDEN;` to all available preprocessor directive rules (see Listing 23). All lexers and parsers generated by ANTLR have the same base class `BaseRecognizer` that owns, among other things, the member `channel` of type integer. The $ sign is ANTLR's way of accessing member variables from within a grammar. A channel represents the connection between a lexer and a parser. A lexer produces tokens and sends them to a channel. A parser reads tokens from a channel and processes them. The default channel for both the lexer and the parser is set to 0. By setting the lexer's channel to a different value than 0 for a given rule, say 99 (`HIDDEN` is a constant for exactly this value), causes the lexer to write the rule's tokens to channel 99. Thus, a parser would never read these tokens when reading channel 0. This channel concept can therefore be an

Listing 24: Nested Preprocessor Directives

```
1  #define COND_A
2
3  #if COND_A
4    a
5    #if COND_B
6      = 0;
7    #elif COND_C
8      = 1;
9    #else
10     =
11   #endif
12   2;
13 #endif
```

effective mechanism for removing irrelevant tokens for the parser. However, there is also a lexer method called `skip` that does not redirect a token via channels, but really skips it to lower memory consumption.

Consider Listing 24. Line 1 contains a macro definition for the macro variable `COND_A`. `COND_B` and `COND_C` are, however, not defined so that the else-case of the conditional block from line 5 to 11 is chosen. Thus, the resulting code produced by a valid preprocessor corresponds to `a=2;`.

This example demonstrates the nested logic that is necessary for implementing conditional preprocessor directives. For this purpose, we choose a stack with the operations push, pop, and peek on boolean values and integrate it into the lexer by means of the action code section `@members`.[8] Each time, the lexer has scanned a condition of an if-directive, we instruct the lexer to evaluate it and then to decide whether the if-body should be transmitted to the default channel or just be skipped. In any case, the evaluated condition is pushed onto the stack.

Turning back to our example in Listing 24, the lexer's stack initially contains a single `true` indicating the preprocessor should not skip characters. In line 1, the lexer detects the macro definition `COND_A` and stores it. When reaching line 3, the lexer recognizes an if-directive and evaluates the condition. Since `COND_A` is defined, the condition evaluates to `true` and is pushed onto the stack. When scanning for

[8]The ANTLR keyword `@members` represents an action code section that is automatically inserted into the generated lexer class after the member declarations.

Listing 25: Ideal If-Directive Rule

```
1 fragment Pp_if_section
2   : WHITESPACE? SHARP WHITESPACE? 'if' WHITESPACE
3       b=Pp_expression Pp_new_line
4       {push(b);}
5   ;
```

Listing 26: Our If-Directive Rule

```
1 fragment Pp_if_section
2 @init {Expression exprObj = new Expression();}
3   : WHITESPACE? SHARP WHITESPACE? 'if' WHITESPACE
      Pp_expression[exprObj] Pp_new_line
4       {push(exprObj.isTrue());}
5   ;
```

the next character, the lexer checks whether the current value on it's stack is true or false. If it is true, the token is passed to the current channel, otherwise it is skipped. When reaching line 5, the lexer detects another if-directive, evaluates it's condition and pushes the resulting boolean value onto the stack. This time, `false` is pushed because `COND_B` is not defined. From now on, every token is skipped until the lexer reaches another preprocessor directive. Each value of the condition from another if-directive of the same conditional block is combined with the current value on top of the stack by applying the boolean *and* operator. This guarantees that exactly one single body per conditional block is used. In our example, the else-case is chosen because none of the two previous bodies are used.

Listing 25 shows the ideal lexer rule for matching C# if-directives. `Pp_if_section` matches if `#if` with a conditional expression ends with a newline character. Then, the lexer pushes the return value from `Pp_expression` of type boolean saved in a temporal variable `b` onto the stack. Unfortunately, lexer rules cannot return values. We may also not use the parser grammar for this purpose because the C# specification prescribes that the preprocessor logic has to be done within the lexical analysis.

Lexer rules may, however, have parameters. Thus, we create and pass an object that holds one single boolean value representing the return value. The corresponding

Listing 27: The Primary_Expression Rule for Preprocessor Expressions

```
1 fragment Pp_primary_expression[Expression exprObj]
2   : (TRUE) => TRUE {exprObj.set(true);}
3   | (FALSE) => FALSE {exprObj.set(false);}
4   | Conditional_symbol {exprObj.set(isDefined(
       $Conditional_symbol.text)); }
5   | '(' Pp_expression[exprObj] ')'
6   ;
```

class can be viewed in Appendix A. Listing 26 shows our ANTLR-conform solution. We first instantiate an `Expression` object whose boolean value is initialized with `false`. Then, we pass it to the `Pp_expression` rule. Finally, we access the potentially updated boolean value by the object's method `isTrue()` and push it onto the stack.

We now have a look at the most important preprocessor rule that is responsible for evaluating the primary preprocessor expressions. Listing 27 illustrates it.

The `Pp_primary_expression` is the bottom-most rule in terms of the evaluation order of preprocessor expressions. When reading the token `TRUE` or `FALSE`, it sets the boolean value of the given rule parameter `exprObj` to `true` or `false`, respectively. Rule parameters are enclosed by square brackets after the rule name. They need to be declared in the way the target programming language specifies it. `(TRUE) =>` represents a syntactic predicate. Since the rule `Conditional_symbol` matches a sequence of arbitrary letters, we need syntactic predicates for `TRUE` and `FALSE` to resolve the ambiguity with regard to macro variable identifiers. If `Conditional_symbol` is successfully scanned, the resulting token text, a macro variable identifier, is accessed by `$Conditional_symbol.text`. For looking up whether or not a particular macro variable is defined in the current context, we define and use the method `isDefined`. It takes the name of the variable and returns `true` if the variable is defined, `false` otherwise.[9] The result is then set to the boolean value of the `exprObj`. The last alternative just passes the parameter to the `Pp_expression` rule.

[9]For this purpose, `isDefined` accesses a set of macro variable identifiers that we have additionally added to the lexer.

Listing 28: Overwriting the Lexer Method `mTokens`

```
1  @Override
2  public void mTokens() throws RecognitionException {
3    if (!ifStack.peek()) {
4      mSkiPped_section_part();
5    } else {
6      super.mTokens();
7    }
8  }
```

Listing 29: The Lexer Rule `SkiPped_section_part`

```
1 fragment SkiPped_section_part
2   : WHITESPACE? SkiPped_characters? NEW_LINE
3   | Pp_directive
4   ;
```

Now we know—by means of our stack—when to produce tokens and when to skip their creation. We only have to instruct the lexer to switch to the corresponding mode in the right situations. For this purpose, we overwrite the lexer method `mTokens` as shown in Listing 28.

The lexer invokes `mTokens` for choosing the next proper lexer rule for the current input sequence. The method effectively searches for the first matching lexer rule in the order as defined in the lexer grammar. In our adapted `mTokens` method, we directly use the lexer rule `SkiPped_section_part` instead when the current value on top of the stack is `false`. Listing 29 demonstrates this lexer rule.

The rule either consumes a sequence of whitespaces, arbitrary non-newline characters, and a newline character or a preprocessor directive to re-evaluate the scanning mode. To avoid token creation, we add the `fragment` keyword to the rule.[10] Since we only want to invoke this rule from within our `mTokens` method, we move it to the bottom of the grammar. All other rules above guarantee that `SkiPped_section_part` will never be chosen.

[10] Alternatively, we can append the action code `skip();` to each alternative.

Listing 30: A More Complicated Example of Left-Recursion

```
1  type
2    : value_type
3    | reference_type
4    | type_parameter
5    | type_unsafe
6    ;
7  value_type
8    : struct_type
9    | enum_type
10   ;
11 struct_type
12   : type_name
13   | simple_type
14   | nullable_type
15   ;
16 nullable_type
17   : non_nullable_value_type
         INTERR
18   ;
19 non_nullable_value_type
20   : type
21   ;
```

Listing 31: The Left-Factored Version of the Rule `type`

```
1  type
2    : (type_name
3      | simple_type
4      | enum_type
5
6      | class_type
7      | interface_type
8      | delegate_type
9
10     | type_parameter
11     | 'void' STAR
12   ) (INTERR | rank_specifier
       | STAR)*
13   ;
```

Changes to the Parser

First, we define the dependency to the lexer by setting the `tokenVocab` attribute in the `options` section to the name of the lexer grammar. Second, we replace all literals by the corresponding tokens, e.g., `'true'` by `TRUE`. For this purpose, we implemented a further program. Third, we resolve left-recursive rules. Although a few of them can be transformed automatically by our corresponding program that we have already used for the lexer rules, most of them have to be edited manually because the left-recursions are located at the fourth or even deeper level of subrule invocations. Listing 30 illustrates an example.

Here, the parser rule `type` is recursively invoked by the rule `non_nullable_value_type`. `type` calls `value_type` that in turn calls `struct_type` that in turn calls `nullable_type` that in turn calls `non_nullable_value_type` that in turn re-

Listing 32: Examples for Incorrect Rule Definitions

```
1  member_access:
2        | primary_expression '.' identifier type_argument_list?
3        | predefined_type '.' identifier type_argument_list?
4        | qualified_alias_member '.' identifier
            type_argument_list?
5        ;
6
7  base_access
8        : BASE DOT IDENTIFIER type_argument_list_opt
9        | BASE OPEN_BRACKET expression_list CLOSE_BRACKET
10       ;
11
12 interface_method_declaration:
13       | attributes? 'new'? return_type identifier
            type_parameter_list_opt '(' formal_parameter_list? ')
            ' type_parameter_constraints_clauses? ';'
14       ;
```

cursively calls `type`. Listing 31 shows our non-recursive version that we obtain by inlining some rules and performing left-factoring multiple times.

Fourth, we optimize the parser rules with the help of the optimization program that we have already used for the lexer rules. Fifth, in case of grammar ambiguity, we introduce syntactic and semantic predicates to decide and prioritize what rule should be chosen.

On the one hand, in some cases resolving ambiguity and distinguishing two syntactically identical rules would require an even more invasive change of the original grammar structure. For this reason, we sometimes use a more general rule to replace two specific ones that previously caused ambiguity. As a result, the parser could now also match input that contradicts the C# specification. Hence, we assume C#-conform syntax as input. This assumption is valid because our transformation is not responsible for checking C# conformance, but just for transforming C# software systems to appropriate KDM instances.

On the other hand, we sometimes introduce new, more specific rule definitions to resolve ambiguity. This, however, does not affect the correctness in the specific cases.

Listing 33: The Original Parser Rule `cast_expression`

```
1 cast_expression
2   : OPEN_PARENS type CLOSE_PARENS unary_expression
3   ;
```

Listing 34: The Annotated `cast_expression`

```
1 cast_expression
2   : OPEN_PARENS type CLOSE_PARENS unary_expression
3     -> ^(CAST_EXPRESSION type unary_expression)
4   ;
```

Finally, we still find some rules that are not correct concerning the specification. Listing 32 shows some examples. In all three cases, the italic text parts were missing and thus were added by us. Perhaps Wischik has used an older version of the C# 4 specification or has not copied but transcribed it.

3.4 Definition of the Abstract Syntax Tree

Now that we have an ANTLR-conform lexer grammar with a preprocessor implementation and an ANTLR-conform parser grammar, we need to define an AST. Since a parser only verifies the syntax, we need a way to specify and produce an AST. In the following, we will build one on the basis of the parser grammar and will therefore considerably change it. For this reason, we copy the pure parser grammar and edit only the copy.

So far, the generated tree represents rather a concrete than an abstract syntax tree. Thus, we use ANTLR's two mechanisms to define node hierarchy and to remove unwanted nodes.

Listing 33 and Listing 34 demonstrate the original and the annotated version of the rule that is responsible for cast expressions.

This example is a very simple and readable one. As we can see, the original rule is not changed in an invasive way. We solely add the line 3 in Listing 34. This rewrite rule makes the imaginary token `CAST_EXPRESSION` to the root node of the current subtree and puts the tokens resulting from the rules `type` and `unary_expression`

Listing 35: The Original Parser Rule `and_expression`

```
1 and_expression
2   : equality_expression
3     ( AMP equality_expression )*
4   ;
```

Listing 36: The Annotated `and_expression`

```
1 and_expression
2   : (e1=equality_expression -> $e1)
3     ( AMP e2=equality_expression -> ^(AMP $and_expression $e2
        ) )*
4   ;
```

under it. Thereby, the tokens `OPEN_PARENS` and `CLOSE_PARENS` are read, but not added to the AST.

Listing 35 and Listing 36 represent a more complex AST creation example. They show the original and, respectively, the annotated version of the rule that is responsible for parsing binary *and* expressions.

The original version of `and_expression` either just forwards to `equality_expression`[11] or it additionally consumes one or more `AMP` tokens (representing the `&` character) with associated `equality_expression` rules. The annotated version behaves as followed: As soon as an `AMP` token is detected (see line 3 in Listing 36), it is set as the root node of the new subtree that consists of the AST , previously created by the `and_expression`, and the associated `equality_expression`. For example, if we parse the expression `7 & 5 & 1`, the resulting AST would look like the one depicted in Figure 3.

In this way, the rule `and_expression` returns either the root token of `equality_expression` or the root token `AMP` with two subtrees as children. The other rewrite rule in line 2 of Listing 36 does effectively nothing. However, ANTLR requires rewrite rules either for none of the terminals and nonterminals of a rule or for all. That is why we have to add it for the first `equality_expression`, too.

[11]The `equality_expression` rule handles equal and unequal expressions and invokes further expressions with lower precedence.

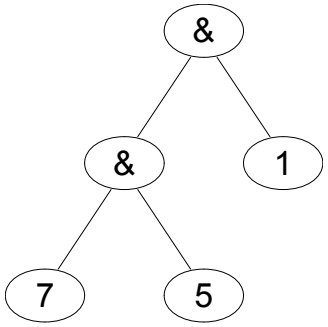


Figure 3: The AST Representation of the Expression `7 & 5 & 1`

4 The Transformation from C# to KDM

In this section, we explain our transformation concept, the architecture of our transformation component, and specific associated problems that we have to solve. Section 4.1 gives an overview of our transformation concept. Subsequently, we describe in Section 4.2 what C# element is mapped to what KDM element. In Section 4.3, we consider it in more detail and especially describe the several transformation phases. Subsequently, we describe the architecture of the transformation component by means of Unified Modeling Language (UML) diagrams in Section 4.4. Finally in Section 4.5, we consider some specific transformation problems and present our corresponding solutions.

4.1 Overview

Figure 4 illustrates the involved transformation parts and steps when transforming a C# system to an appropriate KDM instance.

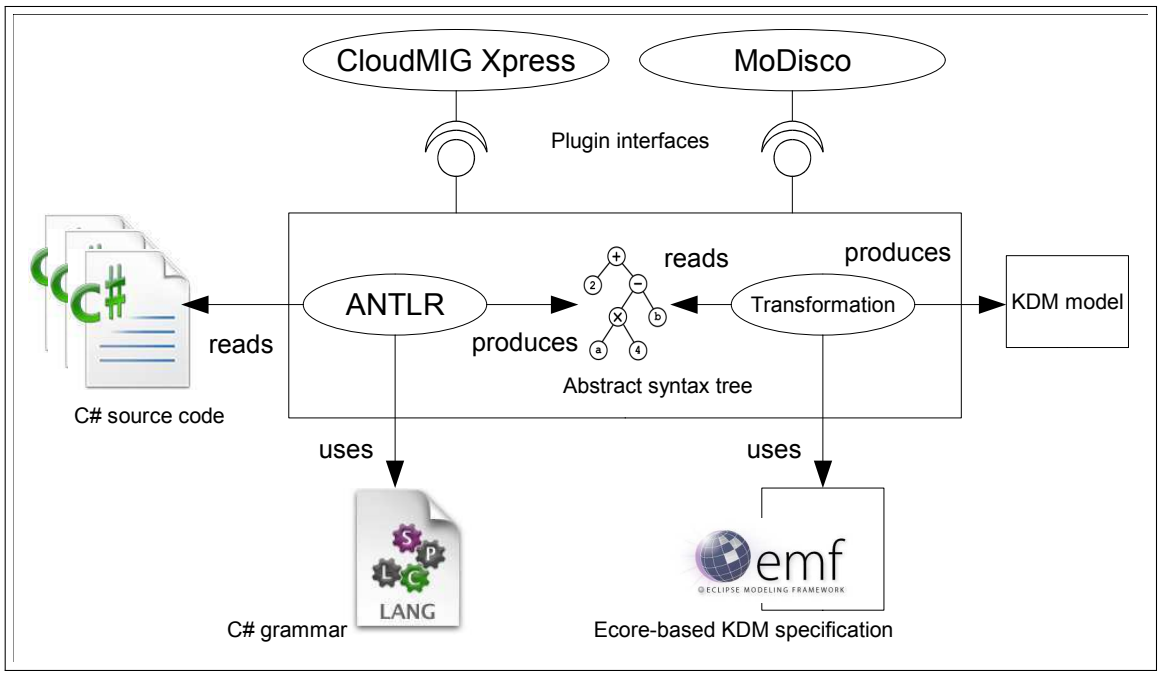


Figure 4: An Overview of Our Transformation Component

We first use ANTLR to parse the system's C# source files and to build the corresponding AST. For this purpose, we use the lexer and parser grammar developed

C# element	**KDM element**
source code file	`CompilationUnit`
namespace	`Namespace`
class	`ClassUnit`
interface	`InterfaceUnit`
struct	`ClassUnit`
enum	`EnumeratedType`
inheritance	`Extends`
implementation	`Implements`
member	`MemberUnit`
static member	`StorableUnit`
method	`MethodUnit`
method parameter	`ParameterUnit`
local variable	`StorableUnit`
statement	`ActionElement`

Table 2: C# to KDM Mapping Table

in Section 3. Then, if the AST is established, we use our Java-based transformation component to map the AST nodes to appropriate KDM elements. Thereby, we use the EMF-based KDM specification to create KDM elements in Java. To be able to integrate our component into CloudMIG Xpress and MoDisco, we also implement the plug-in interfaces that they offer.

4.2 The Mapping from C# to KDM

Before we consider the transformation process in detail, we first have a look at the mapping from C# syntax to KDM model entities. Table 2 presents an overview of the C# and KDM elements that our transformation supports.

Concerning the C# specification, a C# software project consists of several compilation units each of them logically represents a C# source file. A compilation unit owns all types that are defined within the corresponding file. Hence, we add each class, interface, struct, and enum to its compilation unit.

In contrast to Java packages, C# namespaces are not integrated into the logical structure to this extent. One reason for this is that namespaces can be declared beyond directory boundaries. We therefore collect the namespaces in a KDM `Module` called *Namespaces* and append this module to the corresponding code model. The

namespace hierarchy is ensured by adding child elements to the `groupCode` list that each namespace holds.

All kinds of C# statements are mapped to the KDM `ActionElement`. This is due to the fact that we want to be compatible to MoDisco's interpretation of the KDM specification. For more information, we refer to Section 4.5.3.

4.3 Phases of the Transformation

We use three transformation phases P1-P3 to transform a C#-based software system to an appropriate KDM instance. Each phase builds upon the previous one with the exception of the first phase. Figure 5 depicts the phases as UML activity diagram.

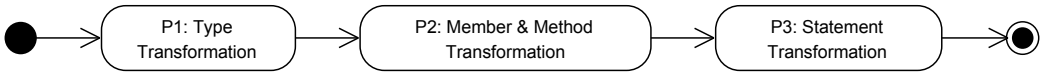


Figure 5: A UML Activity Diagram of the Transformation Phases

4.3.1 Necessity of Three Transformation Phases

Before describing the individual phases in detail, we first want to explain why the use of three phases is necessary. For this purpose, we look at the example C# source file in Listing 37.

Let us assume, we would use a transformation that behaves differently to the one we use. It would try to map the C# source code directly in one single phase by using on-demand type and identifier loading. After parsing the file and building the AST out of it, this transformation would recognize the `CLASS` token with an `IDENTIFIER` token containing the class' name `A`. Then, the transformation would instantiate a new KDM `ClassUnit` element. When reaching the member `z` of type `Zoom`, it would create a KDM `MemberUnit` element of name `z` and type `Zoom`. Let us further assume, the source file that contains the type `Zoom` would be available but not parsed so far and `Zoom` would be located within the same namespace as class `A`. Thus, while mapping the AST of the first file, the transformation would have to look up the current KDM code model for the type `Zoom`. Since this type would not

Listing 37: The Contents of an Example C# Source File

```
1  using System.IO;
2
3  namespace ExampleNamespace {
4    class A {
5      Zoom z;    // Type Zoom is contained in namespace ExampleNamespace
6      File f;    // Type File is contained in namespace System.IO
7
8      void execute() {
9        z = doZoom();
10     }
11
12     Zoom doZoom() {
13       [...]
14     }
15   }
16 }
```

yet be included due to our assumption, the transformation would have to search all source files for it that have not been parsed so far.[12] If it has finally found it, a corresponding KDM type representation could be created, e.g., an `InterfaceUnit`, an `EnumeratedType`, or again a `ClassUnit`. Note that for mapping `Zoom` completely, the transformation would have to potentially repeat the look-up process. In short, the considered transformation would require a recursive on-demand look-up mechanism.

However, this way of processing can lead to problems. First, consider the method `execute` in line 8. It invokes the method `doZoom` whose KDM representation has not yet been built at this point because it is defined later in line 12. To solve this problem, the transformation could behave similar to the on-demand look-up mechanism as described above. In doing so, it additionally needs to save and restore its current position within the current AST. The same applies to class members that are directly initialized at their declarations.

Second, the required logic is intrinsically complex and complicated. In the worst case, for transforming the first parsed source file, we have to load and transform all other source files. In particular, the transformation does not allow to produce an

[12] In contrast to the Java, C# does not require that the name of a source file corresponds to the name of the contained type.

intermediate KDM representation, e.g., one that just contains the KDM elements of the KDM code package without the KDM action package.

Third, introducing efficient parallel processing is almost impossible due to the high degree of dependencies. Although we could use two or more copies of the transformation and distribute them over distinct sets of source files, one needs access to source files of others to resolve type names and other identifiers in general.

Hence, we do not use a single-phase transformation. Instead, we map a C# software system in a three-steps approach that is described in the next three subsections. In this way, we are able to handle the problems mentioned above in a modular and simpler way that is also easier to implement.

As indicated by Figure 5, phase 1 transforms the type definitions. Afterwards, phase 2 handles the mapping from member declarations and method definitions to appropriate KDM elements. Finally, phase 3 transforms the C# statements to the corresponding elements of KDM's action package. In the following, the three phases are described in more detail.

4.3.2 Phase 1: Internal Type Transformation

In the first transformation phase, our transformation component parses all C# source files of the given software system. It utilizes the AST that is produced while parsing to only transform namespaces and type definitions (e.g., classes, interfaces, and structs) with their corresponding modifiers and names. Especially, it intentionally does not transform any inheritance relations, for example, since this would require a look-up mechanism whose disadvantages we want to avoid.

If we again consider Listing 37, our transformation would produce an XML Metadata Interchange (XMI) file with the simplified contents shown in Listing 38. We omit the Extensible Markup Language (XML) attributes of the XML element `Segment` in line 2 and the contents of the inventory model in line 4 for the sake of simplicity.

The KDM instance contains the inventory model (see line 3 to 5) and the code model (see line 6 to 26). The latter owns a KDM `Module` element and a KDM `CompilationUnit` element.

The `Module` element holds all namespaces that are newly defined by the processed source files. Since the example from Listing 37 defines the namespace `Ex-`

Listing 38: The simplified Phase-1 KDM Representation of the Source Code from Listing 37

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <kdm:Segment [...]>
3    <model xsi:type="source:InventoryModel" name="source references">
4    [...]
5    </model>
6    <model xsi:type="code:CodeModel" name="Internal CodeModel">
7      <codeElement xsi:type="code:Module" name="Namespaces">
8        <codeElement xsi:type="code:Namespace" name="global" groupedCode=
             "//@model.1/@codeElement.0/@codeElement.1">
9          <attribute tag="FullyQualifiedName" value=""/>
10       </codeElement>
11       <codeElement xsi:type="code:Namespace" name="ExampleNamespace"
             groupedCode="//@model.1/@codeElement.1/@codeElement.0">
12         <attribute tag="FullyQualifiedName" value="ExampleNamespace"/>
13       </codeElement>
14     </codeElement>
15     <codeElement xsi:type="code:CompilationUnit" name="example.cs">
16       <source>
17         <region file="//@model.0/@inventoryElement.0/@inventoryElement
               .0"/>
18       </source>
19       <codeElement xsi:type="code:ClassUnit" name="A" isAbstract="false
             ">
20         <attribute tag="FullyQualifiedName" value="global.
               ExampleNamespace.A"/>
21         <source>
22           <region startLine="4" startPosition="1"/>
23         </source>
24       </codeElement>
25     </codeElement>
26   </model>
27 </kdm:Segment>
```

`ampleNamespace`, a corresponding KDM `Namespace` element is present in line 11 in Listing 38. The *global* namespace (see line 8) represents the root namespace as defined in the C# specification.

The `CompilationUnit` represents the logical element of the example file. In the first phase, it only holds a reference to the corresponding inventory model element (see line 17) and the file's type definitions. Since the example from Listing 37 defines a class `A`, it is represented by a KDM `ClassUnit` element in line 19 in Listing 38. Additionally, our transformation adds the fully qualified name as a KDM attribute as well as the start row and column of the class definition within the source file.

4.3.3 Phase 2: Internal Member Declarations and Method Definitions Transformation

The second transformation phase is responsible for transforming member declarations and method definitions but again without any member initializers and method bodies.

In the following, we name types *internal types* if they are defined by the considered software system itself, i.e., such types are especially not defined by external libraries. After completing the first phase, the KDM representations of exactly these internal types are available. We call types *external types* if they are defined by foreign or external libraries from which often only the binary executables and not the source code exist on the file system.

Member declarations comprise all declarations at the class level including C# properties, fields, constants, indexer, and events with their corresponding types. Since all internal types are already present in the code model, they are directly accessible and can be referenced. Our transformation has to look up external types only. Moreover, we limit the look-up process to the transformation phases P1 and P2 to reduce complexity. In Section 4.5.1, we describe our whole look-up mechanism in detail.

Let us turn back to the second phase. As mentioned before, our transformation also transforms method definitions in this phase. Apart from the method's name, even its modifiers, return type, type parameters, and formal parameters are mapped to corresponding KDM elements. Listing 39 illustrates another source code example that consists of, among other things, a member `f` and a method `Read`.

Listing 39: Another C# Source Code Example

```
1  using System.IO;
2
3  namespace ExampleNamespace {
4    class B {
5      File f; // Type File is contained in namespace System.IO
6
7      public char Read(string s) {
8        System.Console.WriteLine(s);
9      }
10   }
11 }
```

If we apply our transformation to this source code, we get an XMI file with the contents shown in Listing 40. Here, we again omit some details for the sake of simplicity.

Besides the KDM elements that have resulted from phase 1, we can now also find a KDM `Imports` element in line 14, a KDM `MemberUnit` element in line 20, and a KDM `MethodUnit` element in line 23.

The `Imports` element represents the using directive from line 1 in Listing 39. Its XML attributes `from` and `to` refer to the parent `CompilationUnit` element and, respectively, the `Namespace` element representing `System.IO`.

Since we do not want to mix internal and external types and namespaces, we use another KDM `CodeModel` element to store external KDM representations. That is why the namespaces `System`, `System.IO`, and the class `System.IO.File` is located in the external code model (see line 36 to 51 in Listing 41).

The `MemberUnit` element represents the member `f` of type `File`. Thus, the XML attribute `type` refers to the external type `System.IO.File`. Modifiers are represented by the child KDM attribute element with the tag `export`. As defined in Listing 39, the KDM attribute's `value` of member `f` is set to `none`.

The `MethodUnit` element represents the method `Read`. Its XML `type` attribute refers to the method's child KDM element `Signature` that contains the return type and the formal parameters, representing both as KDM `parameterUnit`s. To differentiate between the return type and the parameters, the return type's XML attribute `kind` is set to `return`. Its `type` attribute refers to the KDM representation of the

Listing 40: The Simplified Phase-2 KDM Representation of the Source Code from Listing 39

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <kdm:Segment [...]>
3    <model xsi:type="source:InventoryModel" name="source references">
4      [...]
5    </model>
6    <model xsi:type="code:CodeModel" name="Internal CodeModel">
7      <codeElement xsi:type="code:Module" name="Namespaces">
8        [...]
9      </codeElement>
10     <codeElement xsi:type="code:CompilationUnit" name="example2.cs">
11       <source>
12         <region file="//@model.0/@inventoryElement.0/@inventoryElement
               .0"/>
13       </source>
14       <codeRelation xsi:type="code:Imports" to="//@model.2/@codeElement
             .0/@codeElement.2" from="//@model.1/@codeElement.0/
             @codeElement.0"/>
15       <codeElement xsi:type="code:ClassUnit" name="B" isAbstract="false
             ">
16         <attribute tag="FullyQualifiedName" value="global.
               ExampleNamespace.B"/>
17         <source>
18           <region startLine="4" startPosition="1"/>
19         </source>
20         <codeElement xsi:type="code:MemberUnit" name="f" type="//@model
               .2/@codeElement.1">
21           <attribute tag="export" value="none "/>
22         </codeElement>
23         <codeElement xsi:type="code:MethodUnit" name="Read" type="//
               @model.1/@codeElement.1/@codeElement.0/@codeElement.1/
               @codeElement.0">
24           <attribute tag="export" value="public "/>
25           <codeElement xsi:type="code:Signature" name="Read">
26             <parameterUnit name="return type" type="//@model.1/
                   @codeElement.3/@codeElement.15" kind="return"/>
27             <parameterUnit name="s" type="//@model.1/@codeElement.3/
                   @codeElement.2" pos="0"/>
28           </codeElement>
29         </codeElement>
30       </codeElement>
31     </codeElement>
32     <codeElement xsi:type="code:LanguageUnit" name="Common C# 
           predefined types">
33     [...]
34     </codeElement>
35   </model>
```

Listing 41: The Simplified Phase-2 KDM Representation of the Source Code from Listing 39 (cont.)

```
36 <model xsi:type="code:CodeModel" name="External␣CodeModel">
37     <codeElement xsi:type="code:Module" name="Namespaces">
38       <codeElement xsi:type="code:Namespace" name="global" groupedCode=
             "//@model.2/@codeElement.0/@codeElement.1">
39         <attribute tag="FullyQualifiedName" value=""/>
40       </codeElement>
41       <codeElement xsi:type="code:Namespace" name="System" groupedCode=
             "//@model.2/@codeElement.0/@codeElement.2">
42         <attribute tag="FullyQualifiedName" value="System"/>
43       </codeElement>
44       <codeElement xsi:type="code:Namespace" name="IO" groupedCode="//
             @model.2/@codeElement.1">
45         <attribute tag="FullyQualifiedName" value="System.IO"/>
46       </codeElement>
47     </codeElement>
48     <codeElement xsi:type="code:ClassUnit" name="File" isAbstract="
         false">
49       <attribute tag="FullyQualifiedName" value="System.IO.File"/>
50     </codeElement>
51   </model>
52 </kdm:Segment>
```

Listing 42: Our KDM Representation of the Common C# Predefined Types

```
<codeElement xsi:type="code:LanguageUnit" name="Common␣C#␣
   predefined␣types">
  <codeElement xsi:type="code:Datatype" name="object"/>
  <codeElement xsi:type="code:Datatype" name="dynamic"/>
  <codeElement xsi:type="code:StringType" name="string"/>
  <codeElement xsi:type="code:OctetType" name="sbyte"/>
  <codeElement xsi:type="code:IntegerType" name="short"/>
  <codeElement xsi:type="code:IntegerType" name="int"/>
  <codeElement xsi:type="code:IntegerType" name="long"/>
  <codeElement xsi:type="code:OctetType" name="byte"/>
  <codeElement xsi:type="code:IntegerType" name="ushort"/>
  <codeElement xsi:type="code:IntegerType" name="uint"/>
  <codeElement xsi:type="code:IntegerType" name="ulong"/>
  <codeElement xsi:type="code:FloatType" name="float"/>
  <codeElement xsi:type="code:FloatType" name="double"/>
  <codeElement xsi:type="code:BooleanType" name="bool"/>
  <codeElement xsi:type="code:CharType" name="char"/>
  <codeElement xsi:type="code:DecimalType" name="decimal"/>
  <codeElement xsi:type="code:VoidType" name="void"/>
</codeElement>
```

C# type `char`. Analogously, the `type` attribute of the parameter `s` points to the KDM representation of C#'s `string` type.

Since primitive types and other types, such as `object` and `string`, are a part of the Common Language Runtime (CLR) and not of any library, they cannot be loaded and transformed from the file system. Thus, we create one KDM `LanguageUnit` element and manually add them to it. Listing 42 demonstrates our `LanguageUnit` element for C# software systems.

4.3.4 Phase 3: Statement Transformation

The final third transformation phase is responsible for mapping C# statements, i.e., especially member initializers and method bodies are transformed. Listing 43 demonstrates the method body of the `Read` method from Listing 39 resulting from the transformation after all three phases.

Listing 43: The KDM Body Representation of the Method `Read` from Listing 39

```
1  <codeElement xsi:type="code:MethodUnit" name="Read" type="//
       @model.1/@codeElement.1/@codeElement.0/@codeElement.1/
       @codeElement.0">
2    [...]
3    <codeElement xsi:type="action:BlockUnit" kind="method␣body"
       >
4      <codeElement xsi:type="action:ActionElement" name="
         expression␣statement" kind="expression␣statement">
5        <codeElement xsi:type="action:ActionElement" name="
           method␣invocation" kind="method␣invocation">
6          <codeElement xsi:type="action:ActionElement" name="
             variable␣access" kind="variable␣access">
7            <actionRelation xsi:type="action:Reads" to="//
               @model.1/@codeElement.1/@codeElement.0/
               @codeElement.1/@codeElement.0/@parameterUnit.1"
               from="//@model.1/@codeElement.1/@codeElement.0/
               @codeElement.1/@codeElement.1/@codeElement.0/
               @codeElement.0"/>
8          </codeElement>
9          <actionRelation xsi:type="action:Calls" to="//@model
             .2/@codeElement.2/@codeElement.0" from="//@model.1/
             @codeElement.1/@codeElement.0/@codeElement.1/
             @codeElement.1/@codeElement.0/@codeElement.0"/>
10       </codeElement>
11     </codeElement>
12   </codeElement>
13 </codeElement>
```

Listing 44: An Excerpt of the External Code Model Resulting from the Transformation of Listing 39

```
1  <model xsi:type="code:CodeModel" name="External␣CodeModel">
2    <codeElement xsi:type="code:Module" name="Namespaces">
3  [...]
4    </codeElement>
5    <codeElement xsi:type="code:ClassUnit" name="File"
       isAbstract="false">
6      <attribute tag="FullyQualifiedName" value="System.IO.
         File"/>
7    </codeElement>
8    <codeElement xsi:type="code:ClassUnit" name="Console"
       isAbstract="false">
9      <attribute tag="FullyQualifiedName" value="System.
         Console"/>
10     <codeElement xsi:type="code:MethodUnit" name="WriteLine
         "/>
11   </codeElement>
12 </model>
```

Since we want to be compatible to MoDisco's implementation of the KDM specification, we adapt its action package conventions. For more information about MoDisco's conventions, we refer to Section 4.5.3.

If we look at line 3, we see a KDM `BlockUnit` that represents the method body. It contains all statements as KDM `ActionElement`s. `System.Console.WriteLine` is not only a method invocation, but also generally an expression statement. Thus, the *method invocation* `ActionElement` in line 5 is embedded in the `ActionElement` *expression statement*. A method invocation `ActionElement` owns an `ActionElement` *variable access* for each argument passed to the method. The target method that is invoked is represented by the KDM `Calls` element. Its XML `from` attribute refers to the *method invocation* `ActionElement`, its `to` attribute points to the `MethodUnit` that represents the method `System.Console.WriteLine`. Since this method is extracted from the .NET framework, it is contained in the external code model as illustrated in line 10 of Listing 44.

Generally, when transforming an external method invocation, our transformation first has to look up the external code model. Then, if it does not find the KDM `MethodUnit`, it searches the file system for the corresponding class and trans-

forms it including the considered method. For more information about the look-up mechanism, we refer to Section 4.5.1.

In conclusion, the three phases introduce a modular transformation strategy that effectively and efficiently implements the mapping from C# source code elements to KDM model elements. In Section 8, we verify our statement by providing accuracy and performance analyses.

Furthermore, our transformation strategy overcomes all three disadvantages that we discussed in Section 4.3.1. First, when transforming statements, all internal types, member declarations, and method definitions are resolved and can be referenced. Second, we simplified the transformation process by introducing three modular transformation phases, each of them is responsible for transforming particular elements. As a result, we also gain two intermediate models, one after the first, the other after the second transformation phase. Finally, our transformation can easily be adapted to run in parallel by parallelizing the individual phases. In each phase, we can use several copies of our transformation that share the work. Since phase 1 requires no look-up, these transformations need not to synchronize themselves. In phase 2 and 3, only the access to the external code model has to be synchronized.

4.4 Architecture of the Transformation Component

In the following, we describe the software architecture of our Java-based transformation component. We begin with considering the five main packages that are also depicted as a UML package diagram in Figure 6.

We adopt the package names *source*, *code*, and *action* from the KDM specification. As Figure 6 indicates, these packages have the same dependency relationships as described in the KDM specification.

The *source* package includes the logic for transforming the physical structure of the given C# project. It especially contains classes and interfaces that are responsible for creating and traversing the inventory model. The *code* package includes everything that is necessary for the implementation of the first two transformation phases P1 and P2. It especially contains transformation helper classes for loop statements, type definitions, operator statements, and member declarations to reduce the file size of the P1-P3 transformation classes. For mapping AST nodes to

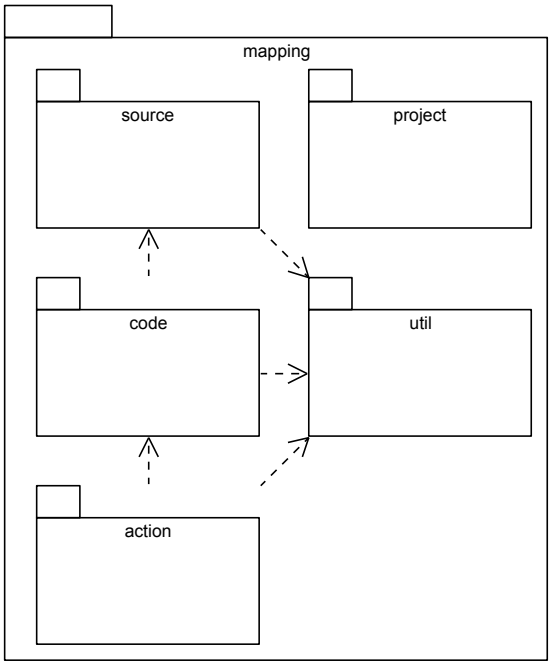


Figure 6: The Packages of Our Transformation Component as UML Package Diagram

corresponding KDM elements, each of these three classes uses a switch-statement. The *code* package, however, also provides classes that are responsible for resolving identifiers and loading external types. The former is described in more detail in Section 4.5.1. The *action* package only contains a small amount of classes for the third transformation phase P3 because most of the logic of the *code* package is also used by this package. We, however, add a class `MicroKDM` consisting of a few MicroKDM string constants that can be used to implement the Micro KDM standard. For more information about the MicroKDM standard, we refer to the Annex A of the KDM specification.

The *util* package contains several helper and utility classes to create and search for KDM elements. Furthermore, it includes a class `ANTLRFileStreamWithBOM` that is able to detect and skip the Byte Order Marker (BOM) of UTF encoded files. We have to implement it by our own because Java does not deliver a file stream that has this feature. Without skipping the BOM, the lexer would complain about unknown characters.

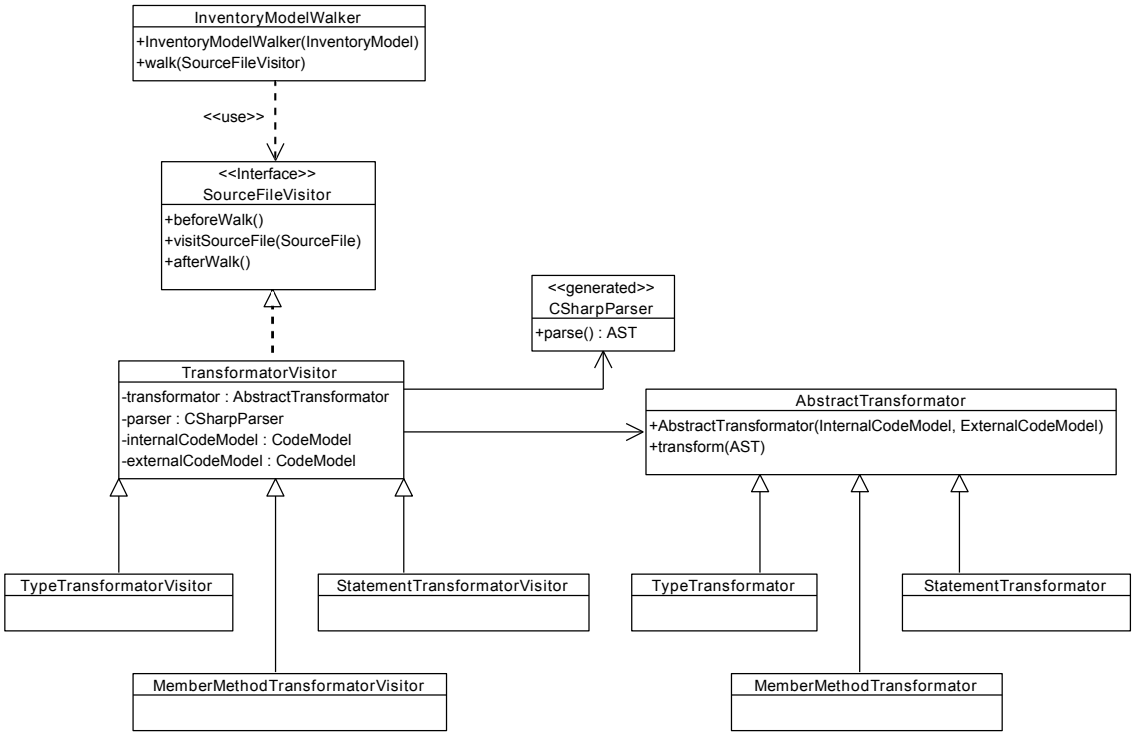


Figure 7: The Core Types of the Transformation Component as UML Class Diagram

The *project* package includes two classes that are responsible for extracting the external library names from a Visual Studio project file. This allows to automatically find and load necessary external types from the file system on demand.

We now have a deeper look at the transformation component. Let us consider the UML class diagram depicted in Figure 7. It shows the classes and interfaces including their associations that are most important for the transformation process.

In the following, we describe what the transformation component does starting from reading the first C# source file of an example C# software system and finishing with the complete KDM instance. For this purpose, we assume that we have already extracted the inventory model of an example C# software system.

For each transformation phase, we use a different transformation class, each of them, however, inherits from the same abstract class `TransformatorVisitor`. This base class implements the interface `SourceFileVisitor` and holds a *transformator* of the type `AbstractTransformator`, a parser of the type `CSharpParser`, and two instances of the KDM type `CodeModel` representing the internal and the external code model.

For the first phase P1, the transformation creates an instance of the `TypeTransformatorVisitor` whose transformator variable is initialized with a new instance of the class `TypeTransformator` passing both a new internal and a new external `CodeModel` to the constructor. This transformator contains the code that transforms C# types to the corresponding KDM elements illustrated in Table 2. The passed code models are also stored in the corresponding members of the `TypeTransformatorVisitor`. The parser is assigned with an instance of the class `CSharpParser` that was generated by ANTLR in advance.

To start the transformation, an instance of the `InventoryModelWalker` is created with the `InventoryModel`. Then, the `walk` method of this instance is invoked with the `TypeTransformatorVisitor` instance from above as argument. In this way, the parser and subsequently the transformator is executed, each time the `InventoryModelWalker` detects a KDM `SourceFile` element of the language C#. More precisely, the transformator's method `transform` receives the AST's root node returned by the parser's method `parse` and transforms the AST nodes that represent type definitions.

For the second and third phase, P2 and P3, respectively, instead of creating an object of `TypeTransformatorVisitor`, an instance of the type `MemberMethodTransformatorVisitor` and, respectively, `StatementTransformatorVisitor` is instantiated. The *transformator* member is initialized by a new instance of `MemberMethodTransformator` and `StatementTransformator`, respectively. The other members are set to the ones passed from the `SourceFileVisitor` of the previous phase.

After completing all three phases, the internal and external code model represent the given C# software system in KDM notation.

4.5 Own and Related Transformation Concepts

Below, we discuss three specific transformation concepts in more detail. The first two deal with resolving types and identifiers. The third concerns MoDisco's interpretation of the KDM specification.

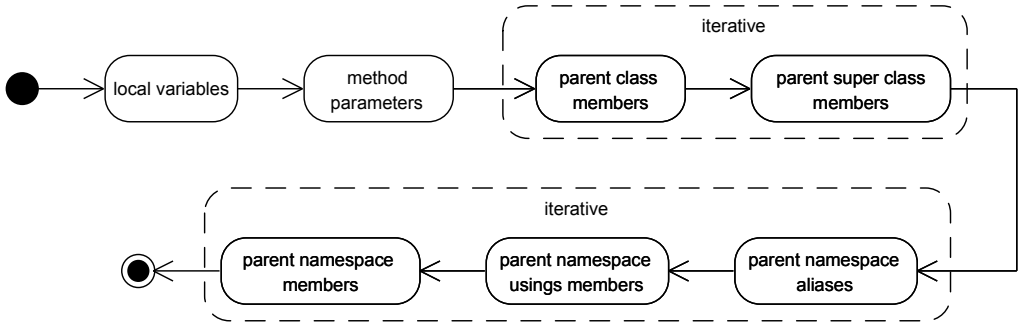


Figure 8: The Name Resolution Approach

4.5.1 Name Resolution

Basically, the goal of our transformation is to transform text to model entities. In our case, we want to transform C# source files to appropriate KDM elements. The latter should especially represent the semantic associations that the whole C# system implies.

Hence, we may not just perform a simple 1-to-1 transformation from the current C# element to the corresponding KDM element. We also have to consider the relationships and other dependencies that exist between different files and folders. In order to resolve such dependencies, it is necessary to navigate through fully qualified names. Transforming the full logical structure of the considered software system is therefore not an easy task.

Below, we present our algorithm to resolve (fully qualified) names. It is applied to find and load internal and external namespaces, types, and methods on the basis of the C# source code.

If our transformation is in phase 2 and recognizes an AST node representing a type, e.g., of a member or parameter declaration, it first searches the internal code model. If it does not find the corresponding type, it searches the external code model. If it still does not find the type, it searches the external libraries.

Figure 8 depicts our full name resolution approach. In phase 2, however, we do not check local variables and method parameters because a type can only be defined as a child of a type or of a namespace.

The look-up process within the internal code model works as follows. First, the requested type is searched in the members of the parent's class because it could have been defined as nested type. Second, it is recursively searched in the inheritance hierarchy of the parent class. After that, it is searched in the set of alias definitions that the parent namespace defines. Then, it is searched within the namespaces that the parent namespace offers by means of its using directives. Subsequently, the requested type is searched in the types that the parent namespace contains. If the type is still not found, the last three steps are repeated for the parent's parent namespace until reaching the global namespace. If the type is not defined in the source code of the software system, it is searched in the external code model.

The look-up process within the external code model works analogous. If the type is still not found in the end, it is searched in the external libraries of the given C# project and then added to the external code model. Since external libraries are often not delivered as C# source code but as precompiled Dynamic Link Library (DLL) files in the Common Intermediate Language (CIL) format, we need a mechanism to search and read such libraries.

4.5.2 Loading External Libraries

We identify two approaches. The first utilizes a CIL-to-C# decompiler to transform a library to its original C# source code. In this way, we can re-use our parser and some parts of our transformation component. The second approach makes use of the C# Reflection API. Below, we discuss both approaches in more detail.

The Decompiler Approach By using a decompiler, we are able to re-use the lexer and parser that are generated from the C# grammar by means of ANTLR. We first let the decompiler decompile all external libraries into one directory and then let our transformation parse them on demand. Most decompilers thereby map the namespace hierarchy to a corresponding directory hierarchy. For the mapping process, we can also re-use some transformation helpers. Although we limit the extraction of external libraries to type definitions, member declarations, and method definitions, it is possible to easily add support for further C# constructs such as getters and setters of properties or particular statements.

One advantage of the decompiler approach is the independence from the .NET framework. The system that performs the transformation does not need the .NET framework to be installed. It only needs the corresponding decompiled source code.

The re-use of already available transformation code turned out to be not as advantageous as we thought at the beginning. In contrast to using the Reflection API, the transformation has to look up the C# artifacts by its own. In particular, finding a nested type is more complex than finding a namespace or a non-nested type because most decompilers[13] only generate one file for each root type. Hence, a nested type is contained in the file of its root type. Moreover, reconstructing the inheritance hierarchy of an external type requires recursively searching in multiple types—and thus potentially in multiple files—as already mentioned in the previous section. That is why we implemented additional, partially complex algorithms that can handle these cases.

When using a decompiler, one should consider the licenses that are delivered with the external libraries. Many licenses, especially the ones of commercial libraries, prohibit the decompilation or restrict it to particular C# artifacts. In such cases, one has to think of the alternative approach via the Reflection API.

Finally, all available decompilers do not always produce syntactically correct C# source code. Most problems occur by generating the special class constructor `.cctor()` and complex statements. On average, we have measured one syntax error per 500 generated files after we have decompiled the .NET 4.0 framework with several different decompilers. Table 3 lists these decompilers and gives additional information about them.

Analysis of Available Decompilers We searched the internet by using Google with the keywords *NET*, *C#*, and *decompiler*. We consider all decompilers that are provided on the first 40 links of this search.[14]

The first column contains the name of the company and the decompiler. The column "License" describes whether the decompiler is free or commercial. For our transformation, we only want to use free decompilers. The third column shows what kind of decompilation formats are supported by the corresponding decompiler.

[13]The rest of available decompilers cannot generate source code at all. More on this later in the next paragraph.

[14]We performed the search on 2011-11-11.

Name	**License**	**Produces**	**Accessable via CL**
Telerik JustDecompile	Free	C# and more	yes
ILSpy	Free	C#	no
DevExtra CodeReflect	Free	C# and more (only in assembly browser)	yes
JetBrains dotPeek	Free	C# (only in assembly browser)	yes
monodis respectively Cecil	Free	CIL	yes
.NET Reflector	Commercial	C#	?
Salamander .NET Decompiler*	Commercial	C# and more	?
DisC# .NET Decompiler*	Commercial	C# and more	?
Spices.Net Decompiler	Commercial	C# and more	?

Table 3: Available C# Decompilers

Since we are only interested in C# source code, we abbreviate other output formats than C# with "and more." Some decompilers can generate but only display the decompiled source code in an assembly browser. In such cases, we append "only in assembly browser" to the output formats. The last column indicates whether the decompiler is executable by means of the command line. This information is especially important if the decompilation should be automatically triggered on demand by the transformation. So far, we decompile all necessary libraries before executing our transformation though.

When we try to decompile the .NET libraries that are required by the applications used in the evaluation in Section 8, all decompilers encounter at least one decompilation error for a particular library. Furthermore, we find syntax errors within multiple decompiled source files. Thus, we use a combination of all free decompilers and additionally adapt the resulting source code files that still contain syntax errors by hand. In doing so, we finally get C#-conform decompilates of the external libraries.

The Reflection Approach An alternative to the decompiler approach is using the CLR's Reflection API[15] similar to Java's Reflection API.[16] For this purpose, we need to write a .NET program that reads a DLL file, extracts the required information (e.g., type name, type member, and type methods) via the Reflection API, and saves the information to a file in an appropriate format (e.g., C# source code or CSV). In this way, the transformation component is able to read the information and use it to build the external code model. Alternatively, the .NET program and the transformation component could communicate via sockets or shared memory.

When using this approach, we do not need to implement, test, and update additional algorithms to resolve external types and methods. The Reflection API internally provides a CIL parser, can look up other dependencies, and thus is able to build an AST that we can use to extract required information. Moreover, since we do not directly generate source code and additionally extract only a limited subset of each library, it is often compatible to common licenses.

One disadvantage is the restriction to the windows operating systems. Although the CIL format represents platform-independent code, it partially contains

[15] `http://msdn.microsoft.com/en-us/library/ms173183%28v=vs.100%29.aspx`
[16] `http://docs.oracle.com/javase/tutorial/reflect/TOC.html`

Listing 45: Modifiers in KDM Notation

```
1 <codeElement xsi:type="code:MemberUnit" name="
     generatedNameNumber" type="id:17" export="public">
2 </codeElement>
```

a .NET wrapper to a DLL that contains platform-dependent code. Furthermore, the CLRs for other operating systems, such as iOS, Solaris, and Unix, do not support the .NET 4 and partially even the .NET 3 framework. Another disadvantage is the fact that we need an interchange format and possibly another programming language.

4.5.3 MoDisco's Approach

As we want to integrate our transformation component into MoDisco, we have to be compatible to MoDisco's custom KDM format. This format is used in MoDisco's Java-To-KDM transformation, for example. Actually, the KDM specification provides specific elements with well-defined semantics, e.g., to represent the control flow and the access to variables within method bodies. Moreover, it recommends several conventions for representing specific operations such as *equals*, *subtract*, and *remove*.[17] For more information concerning recommendations and conventions, we refer to the corresponding section "MicroKDM" in the KDM specification [17, Annex A].

MoDisco mostly uses generic KDM elements with under-specified semantics. To be able to represent the underlying specific programming language constructs, it adds additional model elements and introduces an own semantics.

In the following, we consider two examples of MoDisco's custom KDM implementations. The first one deals with modifiers such as `private` or `static`. Listing 45 and Listing 46 show the notation recommended by the KDM specification and MoDisco's version, respectively.

For representing the modifier of a member variable, the KDM specification proposes the `MemberUnit`'s attribute `export`. In Listing 45, this attribute is set to `public`. MoDisco uses an KDM `Attribute` element with the tag `export` instead.

[17] on list types

Listing 46: Modifiers in MoDisco's Notation

```
1 <codeElement xsi:type="code:MemberUnit" name="
     generatedNameNumber" type="id:17">
2   <attribute tag="export" value="public␣"/>
3 </codeElement>
```

Listing 47: Method Invocation in KDM Notation

```
1 <codeElement xmi:id="id.2" xsi:type="action:BlockUnit" kind="
     Compound">
2   <entryFlow from="id.2" to="id.3" />
3   <codeElement xmi:id="id.3" xmi:type="action:ActionElement"
       kind="MethodCall">
4     <actionRelation xmi:type="action:Reads" to="id.85" from="
        id.3"/>
5     <actionRelation xmi:type="action:Calls" to="id.81" from="
        id.3"/>
6   </codeElement>
7 </codeElement>
```

The value is then set to the modifier. In Listing 46 the export attribute's value is set to `public`.

Another example is illustrated by Listing 47 and Listing 48. Both listings represent the same method call with one actual parameter.

Since KDM is an entity-relationship model, the order of elements has no semantics. For representing order, KDM offers control flow elements. Line 2 in Listing 47 shows the `EntryFlow` element that represents the control flow from the method body (id.2) to the method call (id.3). In line 3, the `ActionElement`'s `kind` attribute is set to `MethodCall` to indicate that this `Actionelement` represents a method call (c.f. the MicroKDM specification [17, Annex A]). The `Reads` relation in line 4 represents the access from the method call (id.3) to the actual parameter (id.85). The latter is not shown in the listing. The `Calls` relation in line 5 represents the target of the method call (id.81). Here, the target is also not shown in the listing.

MoDisco wraps everything in a new `ActionElement`. For this reason, we can also see an `ActionElement` of kind `expression statement` in line 2 in Listing 48. Originally, MoDisco's Java-To-KDM transformation only creates the `Calls` relation

Listing 48: Method Invocation MoDisco's notation

```
1  <codeElement xmi:id="id.2" xsi:type="action:BlockUnit" kind
       ="method␣body">
2    <codeElement xsi:type="action:ActionElement" name="
         expression␣statement" kind="expression␣statement">
3      <codeElement xsi:type="action:ActionElement" name="
           method␣invocation" kind="method␣invocation">
4        <codeElement xmi:id="id.3" xsi:type="action:
             ActionElement" name="variable␣access" kind="
             variable␣access">
5          <actionRelation xsi:type="action:Reads" to="id.85"
               from="id.3"/>
6        </codeElement>
7        <actionRelation xsi:type="action:Calls" to="id.81"
             from="id.3"/>
8      </codeElement>
9    </codeElement>
10 </codeElement>
```

without the `ActionElement` of kind `variable access` from line 4 to 6. In this way, MoDisco is not able to represent method call arguments. That is why we additionally build the `ActionElement` containing a `Reads` relation to the parameter. Furthermore, MoDisco assumes that the order of the elements in the model represents the control flow.

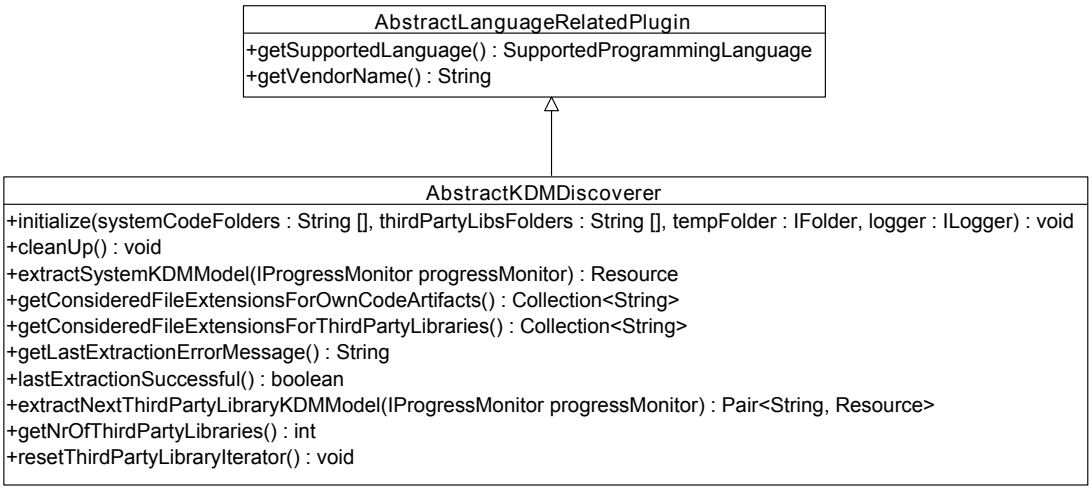


Figure 9: CloudMIG Xpress' Plug-in Interface as UML Class Diagram

5 Integration of the Transformation Component

In this section, we present the plug-in interfaces of CloudMIG Xpress and MoDisco because we want to integrate our transformation into both frameworks. As already mentioned in Section 2.2, CloudMIG Xpress assists in migrating a system to the cloud, whereas MoDisco helps to reconstruct and understand legacy systems by means of model-driven reverse engineering. MoDisco terms a transformation that transforms one language into another one, e.g., Java to KDM, a *discoverer*. Since CloudMIG Xpress adapts this term, we will use it, too.

5.1 Plug-In for CloudMIG Xpress

Figure 9 shows CloudMIG Xpress' plug-in interface as a UML class diagram. An implementation of this interface needs to inherit the abstract class `AbstractKDMDiscoverer` and its super class `AbstractLanguageRelatedPlugin`.

First, we implement the two methods that the class `AbstractLanguageRelatedPlugin` offers. The method `getSupportedLanguage()` returns an enum value of the type `SupportedProgrammingLanguage` that indicates for what programming language the discoverer can be applied. Thus, our implementation returns the enum value `SupportedProgrammingLanguage.CSHARP`. The other method `getVen-`

`dorName()` returns a unique description of the specific transformation component in order to distinguish multiple transformations that are applicable on the same programming language.

Second, we implement the methods provided by the class `AbstractKDMDiscoverer`. Since our transformation loads the individual required types of the external libraries on demand and not in advance, we do not use the interface methods `extractNextThirdPartyLibraryKDMModel()`, `getNrOfThirdPartyLibraries()`, and `resetThirdPartyLibraryIterator()` for handling external libraries.

We begin with implementing the methods `initialize()` and `cleanUp()`. The former is the entry point of the plug-in and receives the folders as parameters that contain the system's source code and the external libraries. Furthermore, a temporary folder and a logger object are passed. Since we do not need to store temporary files or folders for our transformation, we do not use the temporary folder. We, however, use the logger object to display some progress information on CloudMIG Xpress' visual progress list while the transformation is being executed. Within the method `initialize()`, we save the required parameters as fields in order to use them for the transformation later. When the transformation has terminated, there is nothing to clean up. For this reason, we let the method `cleanUp()` empty.

The methods `getConsideredFileExtensionsForOwnCodeArtifacts()` and `getConsideredFileExtensionsForThirdPartyLibraries()` should return the file extensions of the files that an implementation of the plug-in interface is able to process in terms of own code artifacts and external libraries, respectively. Both methods are used to present the CloudMIG Xpress user only those files in the selection dialog that the user needs in order to transform a system of the given language. Hence, we let the former method return a collection containing the file extension `cs` and the latter return an empty collection.

The actual transformation is trigged by CloudMIG Xpress that invokes the method `extractSystemKDMModel()`. The passed `progressMonitor` parameter can be used to give information about the current transformation progress. Each time, one of the folders that are passed as parameters in the method `initialize()` has been processed, the method `lastExtractionSuccessful()` is invoked by CloudMIG Xpress. Its return value indicates whether the transformation was successful or not. If any error occurs while our transformation is being executed, we report it

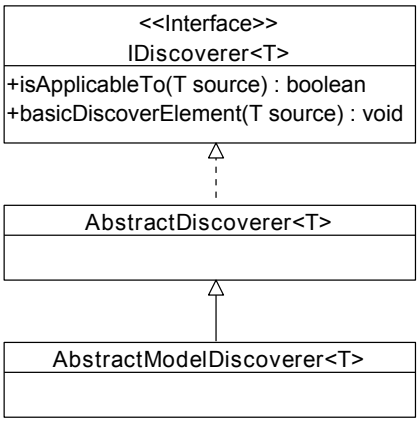


Figure 10: MoDisco's Plug-in Interface as UML Class Diagram

by returning a corresponding error message via the method `getLastExtractionErrorMessage()`.

5.2 Plug-In for MoDisco

Figure 10 shows MoDisco's plug-in interface as a UML class diagram. An implementation of this interface needs to implement the Java interface `IDiscoverer<T>`. MoDisco, however, provides the abstract class `AbstractModelDiscoverer<T>` that already implements basic logic for all kinds of model discoverers, especially for our C#-To-KDM model discoverer. It inherits from the class `AbstractDiscoverer<T>` that in turn implements the interface `IDiscoverer<T>` so that we can use it instead of the interface.

For integrating our transformation into MoDisco, we need to implement the two methods `isApplicableTo()` and `basicDiscoverElement()` that are originally defined in the interface `IDiscoverer<T>`.

Before doing so, we define what kind of type the type parameter of the abstract class `AbstractModelDiscoverer<T>` should take. Since our transformation can operate on files and folders, we implement two versions of it: One that extends `AbstractModelDiscoverer<IFile>` and one that extends `AbstractModelDiscoverer<IFolder>`.

Now, we implement the method `isApplicableTo()`. For the file version, we let the method return true if the given `source` of type `IFile` represents a C# file.

We can check this, e.g., by means of its file extension as it is done by CloudMIG Xpress, too. For the folder version, we let the method always return true.

Finally, we also implement the method `basicDiscoverElement()`. For the file version, we let the method execute our transformation on a single file that is passed as parameter. However, the operating range is limited to files that have no dependencies at all or some that refer to external libraries. Since internal dependencies to other source files of the system are not contained in the decompilation folder, our transformation cannot access them. For the folder version, we let the method execute our transformation on the passed folder. If the transformation fails, the error is reported by the provided `DiscoveryException`.

Moreover, MoDisco requires some coding conventions and standards,[18] e.g., a package naming convention, a source code formatting convention, and the availability of an undefined amount of tests. However, these requirements are only necessary if the plug-in should be contributed to the official MoDisco release.

[18] `http://wiki.eclipse.org/MoDisco/How_to_contribute`

Virtual Machine Size	CPU Cores	Memory	Disk Space in Web and Worker Roles	Disk Space in a VM Role	Allocated Bandwidth (Mbps)	Cost Per Hour ($)
ExtraSmall	Shared	768 MB	19,480 MB	20 GB	5	0.02
Small	1	1.75 GB	229 MB	165 GB	100	0.12
Medium	2	3.5 GB	500 MB	340 GB	200	0.24
Large	4	7 GB	1 GB	850 GB	400	0.48
ExtraLarge	8	14 GB	2 GB	1890 GB	800	0.96

Table 4: Available VM Sizes

6 Microsoft Azure Cloud Profile

In this section, we present the cloud environment model for Microsoft Azure. It, however, represents only the VM role with the characteristics that are necessary to later apply the constraint validator of Section 7. Both the worker role and the web role can be easily added though. We therefore focus on the VM role now. Section 6.1 describes the different available hardware configurations, i.e., the VM instance types that Azure provides. Section 6.2 briefly shows the definition of the transient storage constraint within the cloud profile. The representation of Microsoft Azure's data center locations is described in Section 6.3.

6.1 Hardware Configuration

Microsoft Azure offers five different VM instance types that are illustrated in Table 4. They differ in the number of CPU cores, in memory, in disk space, in the maximal bandwidth, and the cost per hour.

The current cloud profile represents all five VM sizes with all but the pricing information. As an example, the CEM representation of the *ExtraSmall* VM size is shown in Listing 11.

An `hardwareConfiguration` element corresponds to one VM instance type. For each CPU core of a VM size, a `cpu` element with the corresponding clock speed is added to the `hardwareConfiguration` (see line 2 in Listing 11). Analogously, a

```
1 <hardwareConfiguration description="Extra␣Small␣Instance" id
     ="extra.small" name="Extra␣Small␣Instance" maxParallel="20
     " startDelayInSec="0">
2   <cpu frequency="1.8" unit="GHz"/>
3   <memory size="768" unit="MB"/>
4   <networkBandwidth amountPerSecond="5"
       counterpartPartitionID="" unitPerSecond="Mbit"/>
5   <storage size="20" unit="GB"/>
6 </hardwareConfiguration>
```

Figure 11: Azure's *ExtraSmall* VM Size

`memory`, a `networkBandwidth`, and a `storage` element is attached to the configuration (see line 3-5 in Listing 11).

6.2 Constraints

Microsoft recommends to use an Azure drive to store data permanently. On the official website, it is said:[19]

> *You can specify that a local storage resource be preserved when an instance is recycled. However, data that is saved to the local file system of the virtual machine is not guaranteed to be durable. If your role requires durable data, it is recommended that you use a Windows Azure drive to store file data.*

Thus, Azure limits the use of the local file system in so far as writing operations are only useful for caching and not for persisting purposes. For this reason, the cloud profile for Azure includes an constraint definition that exactly represents this limitation. Moreover, it also contains suggestions for persistence storage resources that Azure offers. Listing 12 illustrates the constraint in a simplified way.[20]

6.3 Partitions

Microsoft operates thousands of servers that are distributed among the world. For increased network performance and improved reliability, Azure lets the cloud user

[19] `http://msdn.microsoft.com/en-us/library/windowsazure/ee758708.aspx` (2012-03-26)
[20] The `id` and `possibleFixViaCEConfiguration` attributes are omitted for the sake of simplicity.

```
1 <constraintConfiguration name="Constraints">
2   <constraint xsi:type="constraints:
       LocalTransientStorageConstraint" descr="Microsoft␣Azure's
       ␣VM␣instances␣used␣in␣the␣virtual␣machine␣role␣cannot␣
       store␣local␣data␣persistently." name="Local␣Transient␣
       Storage" [...]>
3     <proposedSolution solution="You␣may␣utilize␣Microsoft␣
         Azure's␣BLOB␣storage␣to␣store␣persistent␣data."/>
4     <proposedSolution solution="You␣may␣utilize␣Microsoft␣
         Azure's␣Table␣storage␣to␣store␣persistent␣data."/>
5     <proposedSolution solution="You␣may␣utilize␣Microsoft␣
         Azure's␣Windows␣Azure␣Drive␣storage␣to␣store␣persistent
         ␣data."/>
6     <proposedSolution solution="You␣may␣utilize␣Microsoft␣
         Azure's␣SQL␣Azure␣storage␣to␣store␣persistent␣data."/>
7   </constraint>
8 </constraintConfiguration>
```

Figure 12: Azure's `LocalTransientStorageConstraint` (simplified)

choose from several data center locations. The meta-model of the cloud profile, CEM, can also represent such different data center locations. Listing 13 shows one `realm` element that comprises the data center locations of North Europe, Western Europe, and North Central US. Each `supportsHWConfiguration` element indicates what hardware configurations are available at the corresponding location. In the example listing, they are omitted for the sake of simplicity.

```
1    <partition xsi:type="iaas:Realm" id="org.cloudmig.
        cloudprofiles.mswindowsazure.realms.zone1" name="Zone␣1
        " arbitraryImages="true">
2      <location id="org.cloudmig.cloudprofiles.mswindowsazure
          .locations.northeurope" name="North␣Europe"
          supportsHWConfiguration="[...]"/>
3      <location id="org.cloudmig.cloudprofiles.mswindowsazure
          .locations.westerneurope" name="Western␣Europe"
          supportsHWConfiguration="[...]"/>
4      <location id="org.cloudmig.cloudprofiles.mswindowsazure
          .locations.northcentralus" name="North␣Central␣US"
          supportsHWConfiguration="[...]"/>
5    </partition>
```

Figure 13: The Representation of Three Data Center Locations of Microsoft Azure (simplified)

7 Microsoft Azure Conformance Checking

This section briefly presents the conformance checking approach of CloudMIG Xpress in conjunction with Microsoft Azure and C#. We then implement one constraint validator that checks for the presence of write accesses to the file system. Other C#-specific validators can be easily added. Here, we only want to give an example of how to plug in a constraint validator to CloudMIG Xpress.

The CloudMIG approach represents a model-driven cloud migration assistant framework. CloudMIG Xpress therefore operates on models that base on the KDM standard. Especially, the conformance checking mechanism uses the extracted architecture model and the extracted utilization model to validate the cloud environment constraints that the chosen cloud provider imposes. For each constraint, one has to implement a validator that at least conforms to the abstract class `AbstractConstraintValidator` that we do not describe here in more detail. In doing so, CloudMIG Xpress is able to apply these validators to search for the corresponding constraints in the KDM models. Some constraints are applicable to all kinds of KDM instances, some are specific to particular programming language libraries and APIs. For the former constraints, CloudMIG Xpress already provides a few validators, e.g., one that checks for a maximal number of files. For the latter constraints, CloudMIG Xpress currently offers some validators for the Java programming language and its APIs. For example, there is one validator that forbids to open sockets and another that prohibits the access to the Reflection API.

We now implement a validator for the C# programming language that checks for write accesses on the local file system. Since stored data that is not saved in one of the different types of Azure's databases is not available anymore after a restart of the VM or a distribution to another VM. CloudMIG Xpress already provides the abstract class `AbstractLocalTransientStorageConstraintValidator` that implements most of the logic that is necessary to detect this constraint within the architecture model. We only need to implement the three methods `isSuitedFor()`, `getForbiddenMethodCalls()`, and `getForbiddenTypesCommonTypeListIDs()`. Figure 14 shows the class `AbstractLocalTransientStorageConstraintValidator` and the super classes providing the methods that are not already implemented by other super classes.

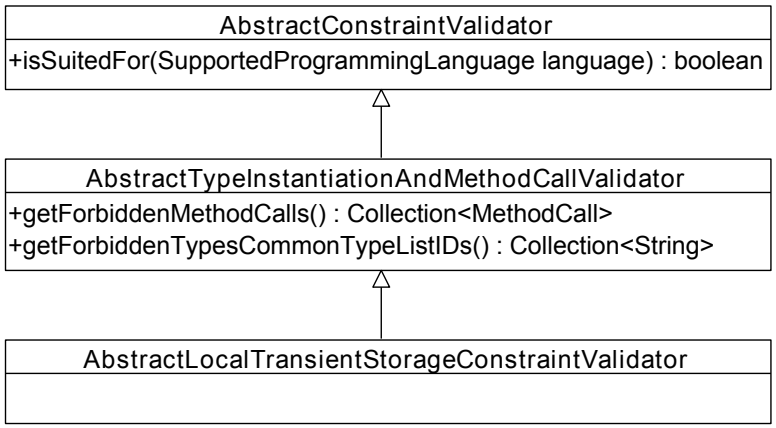


Figure 14: CloudMIG Xpress' Validator Interface for the Transient Storage Constraint as UML Class Diagram

The method `isSuitedFor()` takes one parameter that indicates what programming language was used to build the system. The method should return `true`, if the validator is applicable to the programming language and thus to the system, otherwise `false`. For our validator, we return `true` if and only if the passed programming language equals C#. The methods `getForbiddenMethodCalls()` and `getForbiddenTypesCommonTypeListIDs()` return a collection of method calls and, respectively, type definitions that the cloud provider discourages from using or even forbids to use. The list of C# method calls and the list of C# types that are responsible for writing to the local file system are illustrated in Listing 15 and Listing 16, respectively. Hence, these method calls and type definitions are returned by the aforementioned methods of our validator.

```
1  System.IO.Directory.CreateDirectory
2  System.IO.Directory.Delete
3  System.IO.Directory.Move
4  System.IO.DirectoryInfo.Create
5  System.IO.DirectoryInfo.CreateSubdirectory
6  System.IO.DirectoryInfo.Delete
7  System.IO.DirectoryInfo.MoveTo
8  System.IO.File.AppendAllLines
9  System.IO.File.AppendAllText
10 System.IO.File.AppendText
11 System.IO.File.Copy
12 System.IO.File.Create
13 System.IO.File.CreateText
14 System.IO.File.Delete
15 System.IO.File.Encrypt
16 System.IO.File.Move
17 System.IO.File.OpenWrite
18 System.IO.File.Replace
19 System.IO.File.WriteAllBytes
20 System.IO.File.WriteAllLines
21 System.IO.File.WriteAllText
22 System.IO.FileInfo.AppendText
23 System.IO.FileInfo.CopyTo
24 System.IO.FileInfo.Create
25 System.IO.FileInfo.CreateText
26 System.IO.FileInfo.Delete
27 System.IO.FileInfo.Encrypt
28 System.IO.FileInfo.MoveTo
29 System.IO.FileInfo.OpenWrite
30 System.IO.FileInfo.Replace
```

Figure 15: C# Method Calls That Are Responsible for Writing to the File System

```
1    System.IO.File
2    System.IO.FileInfo
3    System.IO.Directory
4    System.IO.DirectoryInfo
5    System.IO.DriveInfo
6    System.IO.FileStream
7    System.IO.FileSystemInfo
```

Figure 16: C# Types That Are Responsible for Acessing the File System

8 Evaluation

In this section, we evaluate our transformation component in terms of performance and accuracy. Section 8.1 gives an overview of the experimental methodology and presents the applications that are used in the experiments. Section 8.2 describes the planning and execution of the performance analysis. Furthermore, we reason about the scalability of ANTLR and our transformation by analyzing the results. In Section 8.3, we perform a completeness and correctness analysis on our transformation component. Finally, we motivate further experiments in Section 8.4, especially a conformance checking analysis concerning the cloud environment Microsoft Azure. For this purpose, we use the industrial C#-based library *Nordic Analytics* from the HSH Nordbank.

8.1 Overview

In the following, we present our methodology for the experiments. Moreover, we present the applications with whom we perform the analysis.

8.1.1 Methodology

When performing the performance analysis, we measure the program execution time by using `System.currentTimeMillis()`. We take ten measurements per application and use the last nine to compute the median program execution time. We skip the first measurement due to the initial overhead of the JIT compilation.

When performing the completeness analysis, we use the matured C# analysis tool NDepend[21] to compare the number of namespace, type, and method definitions.

8.1.2 Nordic Analytics

Nordic Analytics (HSH Nordbank) is a C#-based library for the assessment and risk control of finance products. It is deployed to a grid infrastructure composed of online trading and batch processing systems. Desktop users have access to it via an Excel front-end.

[21] `http://www.ndepend.com/`

Name	Author	Description	Dependencies	URL	Last Update
SharpDevelop	Christoph Wille	C# IDE with debugger and testing environment	.NET and many more	[a]	2012-03-04
NAnt	Gerry Shaw et. al	.NET build tool	.NET and 3 more	[b]	2011-10-22
RAIL	Bruno Cabral et. al	Runtime Assembly Instrumentation Library for .NET	.NET and 2 more	[c]	2005-01-19

[a] `http://sourceforge.net/projects/sharpdevelop/`
[b] `http://nant.sourceforge.net/`
[c] `http://rail.dei.uc.pt/index.htm`

Table 5: Other Open-Source Applications That We Use for the Evaluations

Nordic Analytics depends on the .NET framework 2.0 and its source code conforms to the C# 4 specification. More information about its physical and logical structure can be retrieved from Table 6.

We use Nordic Analytics as a C#-based industrial example application to analyze and evaluate ANTLR, our transformation component, and CloudMIG's cloud environment constraint violation detection mechanism.

8.1.3 Other Used Applications

Besides Nordic Analytics, we also use other applications for the evaluations. We choose two smaller and one larger C# project to analyze the scalability with sufficient precision. All three applications are released under an open-source license so that we can access and especially transform the source code. Table 5 shows these applications with additional information.

Although the applications do not represent typical examples for a migration to the cloud, they serve as good examples to measure the performance, correctness, and completeness of our transformation component. We do not need particular cloud migration candidates to evaluate the extraction from source code to KDM instances. Here, we intentionally choose the three applications since they represent different orders of magnitude.

8.2 Performance Analysis

Below, we evaluate ANTLR and our transformation with respect to their performance. Section 8.2.1 to Section 8.2.6 describe the goals, the experimental setting, the scenarios, the results, and finally the threats to validity.

8.2.1 Goals

The following experiments evaluate the performance of ANTLR and our transformation component. Furthermore, the experiments analyze whether they scale according to appropriate metrics representing the size of a system.

We expect that ANTLR scales with respect to the file size of a system, i.e., the processing time of C# source files by the generated lexer and parser is linear in time. Moreover, we also expect the processing period of our transformation component to grow linear in time for larger systems.

8.2.2 Experimental Setting

We perform our analysis on a system with an Intel Core2Duo 2x2.4 GHz and 2 GB RAM using the operating system (OS) Windows XP SP3. We disable all command line output messages for the execution of the experiments in order to not falsify the experiment results later. For the same reason, we do not save the KDM instances that are generated by our transformation.

Moreover, we determine the number of files, the lines of code (LOC), the project size,[22] and the number of C# types of the used applications. We need the information for the different scenarios. The number of source files and the project size are given by the OS. The tool *cloc*[23] calculates the LOC for us. We use NDepend to determine the overall number of types that the individual applications define. Table 6 displays the information about each application.

8.2.3 Scenarios

We define five scenarios (S1-S5) for the performance analysis. S1 and S2 serve as a performance and scalability indicator of ANTLR. The last three scenarios (S3-S5)

[22] represents the summarized file size of the project's C# source files

[23] `http://cloc.sourceforge.net/`

Application	**Number of Files**	**LOC**[a]	**Project Size**	**C# Types**[b]
SharpDevelop	6399	618.565	25.44 MB	8.518
NordicAnalytics	939	170.656	11.70 MB	1.355
NAnt	356	43.619	3.42 MB	494
RAIL	36	15.038	695.29 KB	128

[a] w/o comments, w/o blank lines
[b] w/o delegates

Table 6: Basic Information about the Used Applications

deal with the performance analysis of the three phases of our transformation component. In all five scenarios, we measure the program execution time as described previously in Section 8.1.1.

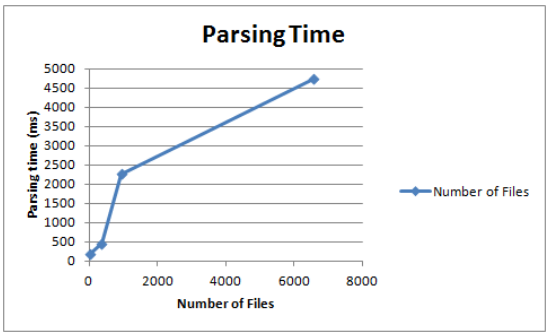
In S1, we let our generated parser parse all C# source files of the given applications presented in Section 8.1.2 and Section 8.1.3. In S2, we let the parser, while parsing, additionally create the AST that we have defined in Section 3.4. In S3, we further execute the first transformation phase. In addition to that, S4 and S5 include the execution of the second and the third transformation phase, respectively.

8.2.4 Results

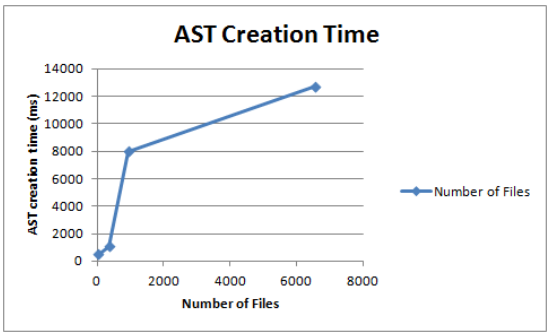
Figure 17a to 21c show the results of the scenarios. Each column of three figures illustrates one scenario. In each case, the y-axis represents the median program execution time. The x-axis represents the number of files, the LOC, and the project size, respectively.

S1 Figure 17a shows the measuring points of *RAIL*, *NAnt*, *Nordic Analytics*, and *SharpDevelop* from left to right. We measured a program execution time of 186 ms, 453 ms, 2,265 ms, and 4,725 ms, respectively, for the corresponding applications. The resulting curve grows non-linearly according to the number of files.

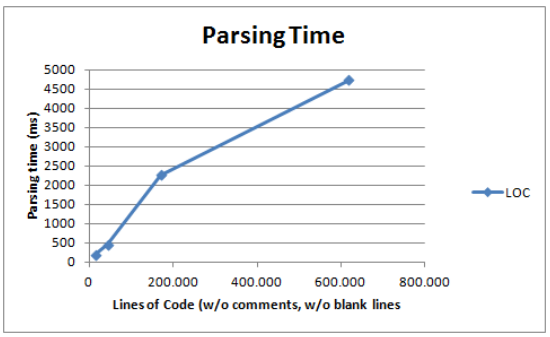
Considering the parsing time in combination with the LOC in Figure 17b, the corresponding curve also grows non-linearly, but not as strong as in the previous figure. If we look at Figure 17c though, we can observe an approximately linearly growing curve.



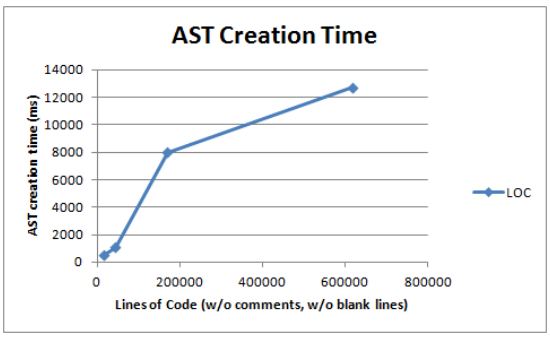
(a) Measured Parsing Time in Relation to the Number of Files



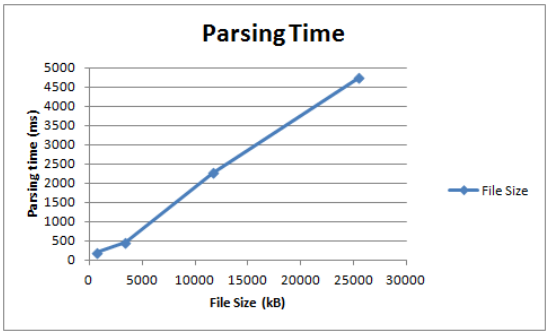
(a) Measured AST Creation Time in Relation to the Number of Files



(b) Measured Parsing Time in Relation to the Lines of Code

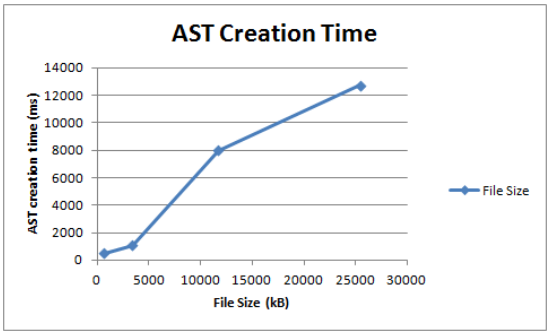


(b) Measured AST Creation Time in Relation to the Lines of Code



(c) Measured Parsing Time in Relation to the Project Size

Figure 17: Scenario 1



(c) Measured AST Creation Time in Relation to the Project Size

Figure 18: Scenario 2

S2 We measured 520 ms, 1,073 ms, 7,993 ms, and 12,707 ms for the AST creation times of the corresponding applications. The resulting curves, illustrated in Figure 18a to 18c, almost exactly correspond to the ones gained by the parsing time. The AST creation takes 2.85 times longer than the parsing on average.

S3 For the program execution time of the P1 transformation, we measured 690 ms, 1,264 ms, 8,285 ms, and 13,091 ms, respectively, resulting in similar curves concerning the previous AST creation time curves. The P1 transformation takes 1.14 times longer than the AST creation on average.

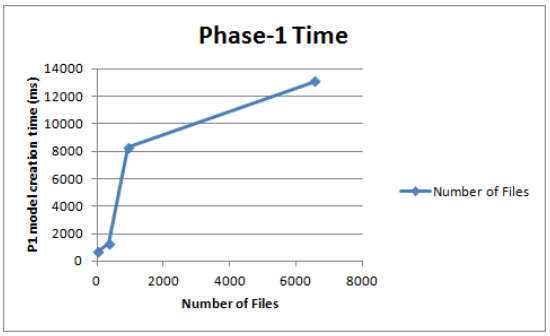
S4 We measured 4,902 ms, 14,363 ms, 53,481 ms, and 207,870 ms for the program execution time of the P2 transformation. The curves in Figure 20a and Figure 20b grow linearly now. Figure 20c shows a curve with a linear growth if we consider the first three measurement points. The program execution time for the P2 transformation of SharpDevelop increased by a factor of more than 15 concerning the P1 transformation. The average time factor that the applications took longer according to their P1 transformation execution times is 10.2.

S5 The execution of the P3 transformation resulted in the following execution times for the corresponding applications: 6,181 ms, 33,740 ms, and 80,285 ms. We have aborted the P3 transformation for SharpDevelop because the first measurement has not terminated after more than 15 minutes. All resulting curves, shown in Figure 21a to 21c, represent an approximately straight line. The P3 transformations took 1.7 times longer than the P2 transformation on average.

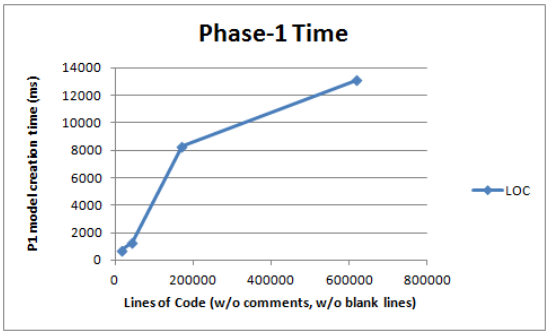
8.2.5 Discussion of the Results

If we consider the results of the first two scenarios, we conclude by means of Figure 17c and Figure 18c that the parsing and the AST creation of ANTLR scales well for increasing input, i.e., project size. Since the metrics *number of files* and *lines of code* do not appropriately represent the input size, the corresponding curves are not that linear as the ones in Figure 17c and Figure 18c.

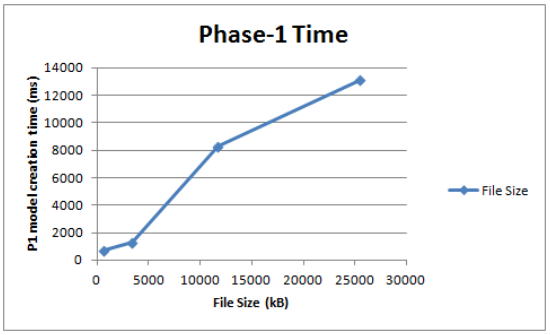
Since the P1 transformation only transforms type definitions, the increase in time compared to the AST creation scenario is not that large (factor of 1.14). Fur-



(a) Measured P1 Time in Relation to the Number of Files

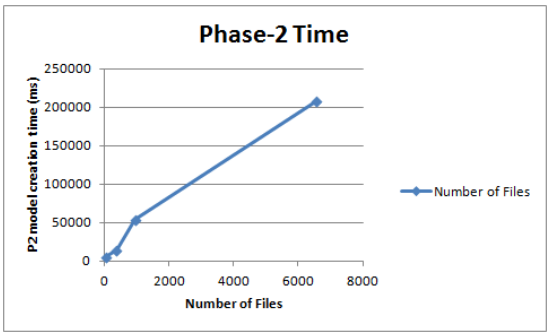


(b) Measured P1 Time in Relation to the Lines of Code

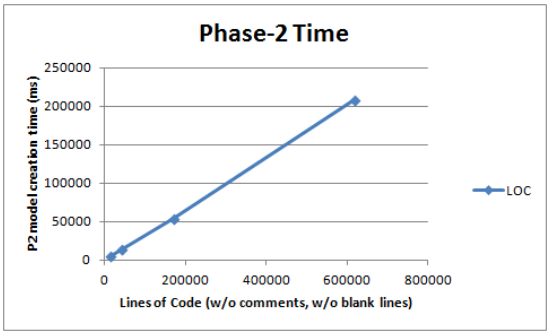


(c) Measured P1 Time in Relation to the Project Size

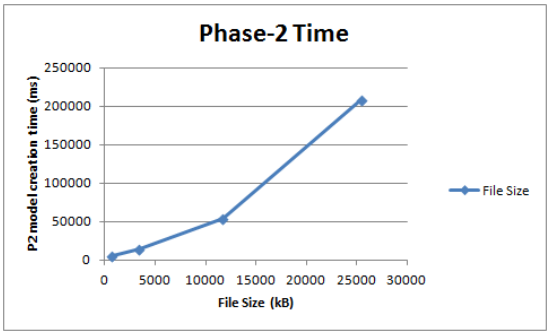
Figure 19: Scenario 3



(a) Measured P2 Time in Relation to the Number of Files

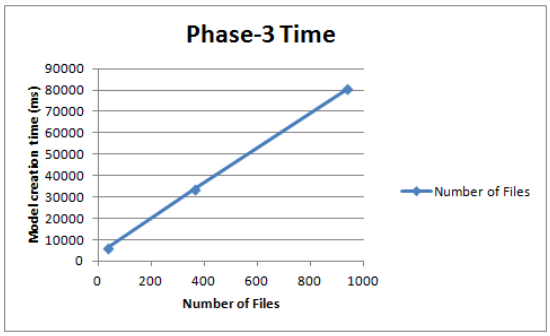


(b) Measured P2 Time in Relation to the Lines of Code

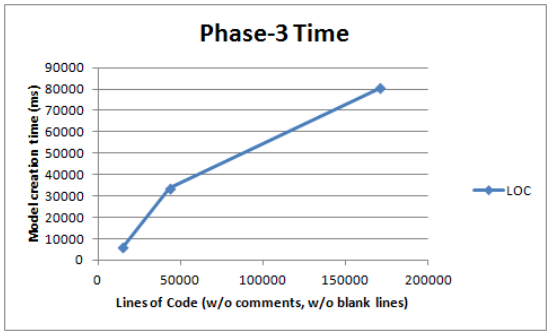


(c) Measured P2 Time in Relation to the Project Size

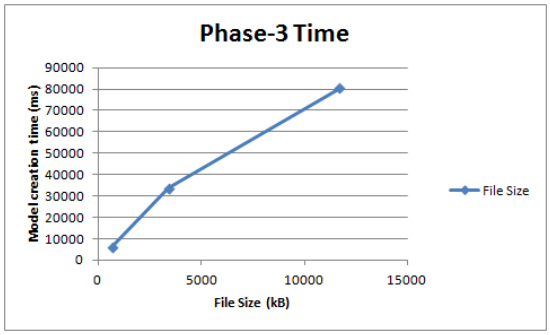
Figure 20: Scenario 4



(a) Measured P3 Time in Relation to the Number of Files



(b) Measured P3 Time in Relation to the Lines of Code



(c) Measured P3 Time in Relation to the Project Size

Figure 21: Scenario 5

thermore, each source file often contains only one or two type definitions on average so that the P1 transformation curves resemble the AST creation curves.

The P2 transformation of all applications took considerably more time than the P1 transformation because it transforms type relationships, method definitions including their parameters, and member declarations inclusively their types. In particular, the name resolution algorithm from Section 4.5.1 is applied to find the correct internal and external types. That is why the P2 transformation required 10 times longer than the P1 transformation. The P2 transformation on SharpDevelop took even longer because we have not decompiled all required external libraries. For this reason, the name resolution algorithm did not terminate prematurely due to a positive match and thus did not find several external types ultimately. This fact results in an abnormally high program execution time. However, Figure 20c shows a linear curve concerning the first three applications. Hence, we conclude that the P2 transformation scales if all external libraries are available.

Due to the missing external libraries, the P3 transformation on SharpDevelop took so long that we finally aborted the process. Therefore, Figure 21a to 21c only display the measurement points of *RAIL*, *NAnt*, and *Nordic Analytics*.

The resulting curve in Figure 21c indicates that even the P3 transformation scales well concerning the project size. We, however, claim that the linear behavior in time depicted by Figure 21a and Figure 21b cannot be generalized to all projects because both metrics do not represent the logical structure and complexity. For more information about this and other threats to validity, we refer to the corresponding Section 8.2.6.

8.2.6 Threats to Validity

Our performance experiments only constitute a first evaluation of our C# grammar (and the generated lexer, parser, and AST) and especially of our transformation component. We do not claim that our evaluation empirically proves the scalability of them. It represents, however, a first positive indicator. To increase the validity, we need to evaluate more applications to check the conclusions we made. Moreover, the project size metric is not optimal since it does not represent the logical structure, e.g., the number of method definitions and the amount of statements. Finally, our transformation component still does not support all C# constructs at the time of

writing. Thus, the final execution times of the P1-P3 transformations will increase in the future.

8.3 Accuracy Analysis

Besides a performance analysis, we also investigate the accuracy of our transformation component. We begin by briefly considering the correctness. Afterwards, we check the completeness of the KDM instance that results from the transformation of *Nordic Analytics*.

Correctness Analysis Testing the correctness of the transformation is an important task since the output need not correspond to the input in general. The more logic we add to our program, the more the risk of bugs increases. As we especially introduce three different, partially complex phases, we should perform an intensive correctness analysis to improve the reliability of and the trust in our transformation component.

However, checking the correctness is very time-consuming. For this reason, we limited our verification to manual code reviews and testing within the context of this thesis. Currently, we provide several JUnit tests that cover the transformation for type definitions, member declarations, method definitions, and a set of C# statements.[24]

We propose an intensive and statistically significant T-Test for future work. Furthermore, simple correctness tests that could be automated should be performed by a program.

Completeness Analysis In the following, we perform a completeness analysis by means of the industrial C# project *Nordic Analytics*. First, we name the goals and describe the experimental setting as well as the scenario. Then, we consider and discuss the result. Finally, we look at potential threats to validity.

8.3.1 Goals

The completeness analysis should reveal to what extent our transformation component transforms all C# constructs of a software system to their corresponding KDM

[24]The full set of JUnit tests can be found on the CD in the annex.

Entity	NDepend	Our Transformation
Namespaces	109	113 (+4, +2.7 %)
Types	1.355	1359 (+4, +0.3 %)
Methods	9.327	6250 (-3077, -67.0 %)
Members	7.169	8.356 (+1187, +16.6 %)

Table 7: Completeness Analysis of Nordic Analytics

elements. In the context of this thesis, we focus on the number of C# namespaces, types, methods, and members for now. We expect from our transformation that it completely transforms these C# constructs with the exception of the number of methods because our transformation does currently not support interface method definitions. Future work will deal with the verification of other constructs such as the type relationships and statements.

8.3.2 Experimental Setting

In order to compare the results of our transformation component, we use the matured .NET code quality analysis tool *NDepend*.

8.3.3 Scenarios

We execute our transformation component on Nordic Analytics and compare the number of namespaces, type definitions (without delegates), method definitions, and member declarations (without constants) with the ones determined by NDepend.

8.3.4 Results

Table 7 shows the analysis and transformation results of NDepend (first column) and our transformation component (second column), respectively. The rows represent the corresponding number of namespaces, types, methods, and members.

We observe that our transformation component transforms almost the same number of namespaces and types as NDepend has analyzed. However, NDepend recognizes more methods, but less members than our transformation component.

8.3.5 Discussion of the Results

We see that our transformation meets our expectations for namespaces and types. The four additional namespaces and types represent dead code, i.e., they are not used by the system. NDepend does not analyze unused elements per default and thus does not consider them in the analysis. The difference in the number of methods results from both the missing support for interface method definitions and from the fact that our transformation maps a C# property not to a getter and/or setter `MethodUnit` but to one single `MemberUnit` for now. The difference in the number of members results from both the property mapping to `MemberUnit`s and the lack of support for constant members.

8.3.6 Threats to Validity

In our completeness analysis, we only compare the amount of entities and do not simultaneously check the correctness. Thus, we could have missed to transform some C# constructs and added a few imaginary ones instead so that we would still consider the transformation to be complete. Moreover, since NDepend skips analyzing dead code per default, the tool could make further unknown assumptions that threatens the validity of the experiment.

8.4 Conformance Checking Analysis

Although our transformation component does not yet support all C# constructs, it is already possible to use it in combination with the cloud profile of Microsoft Azure and the local transient storage constraint validator for C# to perform a first conformance checking analysis on an arbitrary C# software system.

8.4.1 Goals

We evaluate whether CloudMIG Xpress is able to detect too less (false negatives), too many (false positives), or exactly the amount of constraint violations that the given system contains.

8.4.2 Experimental Setting

In order to compare the results of the validator, we use the matured .NET code quality analysis tool *NDepend*.

8.4.3 Scenarios

We execute CloudMIG Xpress and let our validator from Section 7 validate the conformance of *Nordic Analytics* to Azure's local transient storage constraint. We start NDepend and execute the query[25] shown in Listing 49, selecting all forbidden method calls that we also have defined in our validator.

The commented lines indicate forbidden method calls that NDepend is not able to identify. If we would uncomment them, the query would be invalid and could not be executed. Thus, we need to remove these method calls from the query.

8.4.4 Results

Listing 50 displays the methods that contain forbidden method calls according to our validator. It detected seven different methods in four different classes.

Listing 51 shows the method calls that NDepend has detected. In particular, they include all method calls that our validator was able to detect. We sorted the results for the sake of clarity. From line 9 to 13, we listed the additional method calls that CloudMIG Xpress has not recognized.

8.4.5 Discussion of the Results

Comparing the results of CloudMIG Xpress and NDepend, CloudMIG Xpress did not recognize five of twelve forbidden method calls resulting in an overall detection rate of approximately 58 %. If we look at the method calls that exclusively NDepend was able to recognize, we see that all these method calls are not contained in *Nordic Analytics*' KDM instance at all. The reason for this lies in our transformation component that does not yet support all C# statements. As soon as one statement of a method body is not supported, the whole method body is not transformed and

[25] NDepend provides the so-called Code Query Language (CQL) that allows to query the code structure of any .NET application. For more information, we refer to `http://www.ndepend.com/CQL.htm`.

Listing 49: The CQL Query That Selects All Forbidden Method Calls

```
SELECT METHODS WHERE ! IsSpecialName AND
  (
IsDirectlyUsing "System.IO.Directory.CreateDirectory(String)" OR
IsDirectlyUsing "System.IO.Directory.Delete(String,Boolean)" OR
//IsDirectlyUsing "System.IO.Directory.Move(String,String)" OR
IsDirectlyUsing "System.IO.DirectoryInfo.Create()" OR
IsDirectlyUsing "System.IO.DirectoryInfo.CreateSubdirectory(String)" OR
IsDirectlyUsing "System.IO.DirectoryInfo.Delete(Boolean)" OR
//IsDirectlyUsing "System.IO.DirectoryInfo.MoveTo(String)" OR
//IsDirectlyUsing "System.IO.File.AppendAllLines(String,IEnumerable<String>,
    Encoding)" OR
//IsDirectlyUsing "System.IO.File.AppendAllText(String,String,Encoding)" OR
//IsDirectlyUsing "System.IO.File.AppendText(String)" OR
//IsDirectlyUsing "System.IO.File.Copy(String,String,Boolean)" OR
//IsDirectlyUsing "System.IO.File.Create(String,Int32,FileOptions,FileSecurity)"
    OR
//IsDirectlyUsing "System.IO.File.CreateText(String)" OR
IsDirectlyUsing "System.IO.File.Delete(String)" OR
//IsDirectlyUsing "System.IO.File.Encrypt(String)" OR
IsDirectlyUsing "System.IO.File.Move(String,String)" OR
//IsDirectlyUsing "System.IO.File.OpenWrite(String)" OR
//IsDirectlyUsing "System.IO.File.Replace(String,String,String,Boolean)" OR
//IsDirectlyUsing "System.IO.File.WriteAllBytes(String,Byte[])" OR
//IsDirectlyUsing "System.IO.File.WriteAllLines(String,IEnumerable<String>,
    Encoding)" OR
//IsDirectlyUsing "System.IO.File.WriteAllText(String,String,Encoding)" OR
//IsDirectlyUsing "System.IO.FileInfo.AppendText()" OR
IsDirectlyUsing "System.IO.FileInfo.CopyTo(String)" OR
IsDirectlyUsing "System.IO.FileInfo.CopyTo(String,Boolean)" OR
//IsDirectlyUsing "System.IO.FileInfo.Create()" OR
//IsDirectlyUsing "System.IO.FileInfo.CreateText()" OR
//IsDirectlyUsing "System.IO.FileInfo.Delete()" OR
//IsDirectlyUsing "System.IO.FileInfo.Encrypt()" OR
IsDirectlyUsing "System.IO.FileInfo.MoveTo(String)"
//IsDirectlyUsing "System.IO.FileInfo.OpenWrite()" OR
//IsDirectlyUsing "System.IO.FileInfo.Replace(String,String,Boolean)"
)
```

Listing 50: Methods That Were Detected by Our Validator

```
SpecialTask.cs: cSpecialTask.GenerateTestFile()
cAuxTools.cs: cAuxTools.GetOutputPath()
Directory.cs: cDirectory.Create()
ConsoleApp.cs: cConsoleApp.GetResultRootDirectory()
ConsoleApp.cs: cConsoleApp.CopyChangedBatches()
Directory.cs: cDirectory.Delete()
ConsoleApp.cs: cConsoleApp.ClearLogFiles()
```

Listing 51: Methods That Were Detected by NDepend

```
1  SpecialTask.cs: cSpecialTask.GenerateTestFile()
2  cAuxTools.cs: cAuxTools.GetOutputPath()
3  Directory.cs: cDirectory.Create()
4  ConsoleApp.cs: cConsoleApp.GetResultRootDirectory()
5  ConsoleApp.cs: cConsoleApp.CopyChangedBatches()
6  Directory.cs: cDirectory.Delete()
7  ConsoleApp.cs: cConsoleApp.ClearLogFiles()
8
9  File.cs: cFile.WriteStream()
10 CopyDirectory.cs: cCopyDirectory.CopyDirectory()
11 ConsoleApp.cs: cConsoleApp.CopyNonWorkingBatches()
12 EvaluationErrorHandler.cs: cEvaluationErrorHandler.
      RemoveOldProtocols()
13 TestfileGenerator.cs: cTestfileGenerator.Generate()
```

Listing 52: The Method `cAuxTools.GetOutputPath()`

```
1     internal string GetOutputPath(string PathName)
2     {
3         string pathUp = Path.GetDirectoryName(PathName);
4         string outputPath = pathUp + '\\' + "Results";
5         DirectoryInfo op = new DirectoryInfo(outputPath);
6         op.Create();
7         return outputPath;
8     }
```

thus remains empty. We will improve the handling of unsupported statements in future work so that only the unsupported statement is not transformed.

In order to understand and verify the results that both CloudMIG Xpress and NDepend have produced, we exemplarily look at the methods `cAuxTools.GetOutputPath()` and `cConsoleApp.GetResultRootDirectory()`.

The former is illustrated in Listing 52. In line 6, the method `Create()` of the type `DirectoryInfo` is called. Since we defined the invocation of this method as forbidden, CloudMIG Xpress correctly complains about the caller method `cAuxTools.GetOutputPath()`.

The method `cConsoleApp.GetResultRootDirectory()` is shown in Listing 53. If we look at line 14, we can see a call to the method `Delete` of the member `Info` of

Listing 53: The Method `cConsoleApp.GetResultRootDirectory()`

```
1     private cDirectory GetResultRootDirectory()
2     {
3       cDirectory R = m_Root.GetChildDirectory(sResults);
4       if(R.Exists)
5       {
6         if(Is(m_ArgFile))
7         {
8           Write(true);
9           Write("WARNING:␣Result␣directory", true);
10          Write(R.FullName, true);
11          Write("exists␣already.␣It␣will␣be␣overriden.", true
                );
12          Write(true);
13        }
14        R.Info.Delete(true);
15      }
16      return R;
17    }
```

Listing 54: The Class `cDirectory` (excerpt)

```
1   public class cDirectory : INDef
2   {
3     [...]
4     public DirectoryInfo Info { get { return m_DirInfo; } }
5     [...]
6   }
```

the type `cDirectory`. Listing 54 demonstrates an excerpt of the type `cDirectory`. In line 4, we find the property `Info` of the type `DirectoryInfo`. Hence, the method call in line 14 in Listing 53 represents a forbidden method call, too.

The conformance checking analysis therefore shows that CloudMIG Xpress is able to detect forbidden method calls if the underlying KDM instance contains enough information.

8.4.6 Threats to Validity

Since NDepend does not accept all method calls in the query shown in Listing 49, *Nordic Analytics* could invoke these not considered method calls. The overall detection rate of CloudMIG Xpress would then decrease further. However, as already mentioned above, all method calls that exclusively NDepend has detected were not contained in the given KDM instance at all. Thus, it is impossible for CloudMIG Xpress to detect all method calls.

9 Related Work

Since cloud computing and cloud migration are relatively new topics and still emerging, there are only limited cloud migration approaches so far. Most of them [18, 38] only cover a part of the whole migration process, e.g., cost estimation, risk assessment, effort estimation, and decision support in general.

Besides Frey and Hasselbring [14], Peddigari [30] and Venugopal et al. [37] also propose cloud migration frameworks, for instance, but they are still in the planning stage and thus described imprecisely or focus on one specific kind of software systems, respectively. Up to now, researchers often investigate the cloud migration process by performing several migration case studies [3, 35]. For this reason, tool support is also restricted.

Apart from ANTLR, there are a few other parser generator tools available. XText [8, 10], for example, is a framework for developing arbitrary languages such as domain-specific and general purpose languages. In addition to the generation of a lexer and a parser in the target language, it also generates a full Eclipse-based IDE with syntax highlighting, code completion, and quick fixes for the given XText grammar.

Moonen [24] developed MANGROVE, a generator for source model extractors based on island grammars. Island grammars are more flexible than standard grammars because they can also handle syntax errors without aborting the parsing. They have at least one fall-back rule that catches all cases that are not covered by the defined grammar rules. In this way, one can use an island grammar to extract only the information that is necessary for the specific use case. Thus, island grammars are particularly useful for reverse engineering purposes such as model extraction from source code. ANTLR's grammar format also supports fall-back rules and parsing based on an incomplete language definition.

Originally, lexers and parsers are manually and independently implemented. In 1975, Yacc[16] was proposed. It represents a compiler-compiler, i.e., a parser generator. By using a lexer generator such as Lex[20], it was then possible to generate a lexer and a parser from the lexer grammar and the parser grammar. However, both grammar formats are different. Furthermore, the generators originally supported only the target language C.
An overview of textual language tools is given by Merkle [23].

10 Conclusions & Future Work

In this section, we conclude the thesis and present future work that is related to our transformation component, to the Microsoft Azure cloud profile, and to C# constraint validators in general.

10.1 Conclusions

In this thesis, we briefly described the CloudMIG approach of Frey and Hasselbring and their corresponding prototype implementation CloudMIG Xpress. For the latter, we implemented a C#-to-KDM transformation component that bases on a C# grammar in ANTLR notation. During the development, we analyzed several available C# grammars and adapted the chosen one. Furthermore, we discussed some particular transformation concepts and thereby analyzed currently available CIL-to-C# decompilers.

As part of the thesis, we additionally built a cloud profile for the cloud environment Microsoft Azure. By means of the exemplary C# constraint validator that we also developed, it is now possible to check an arbitrary C#-based software system for write accesses on the local file system. In conclusion, we built and embedded the basic structure for CloudMIG Xpress to perform a conformance analysis on any C# software system for the cloud environment Microsoft Azure. Tests and first experiments show that the transformation is complete, correct, and scales well. Furthermore, CloudMIG Xpress was applied on an industrial example application the first time.

10.2 Future Work

There are quite a few places where we can improve our transformation component. First, we will add support for more C# constructs, e.g., for delegates, type parameters, and attributes. Second, we will add support for the alternative look-up approach that utilizes the .NET Reflection API. In addition to that, we will provide a switch mechanism so that the user is able to choose between both. Third, we will automate the decompilation approach so that we do not need to decompile the external libraries in advance. Fourth, we will increase the performance of the transformation by parallelizing the individual phases with the help of threads and

by reducing the amount of parser passes. Currently, a system is parsed three times, i.e, once per phase. Fifth, the C# grammar can be optimized in terms of speed by using more sophisticated syntax predicates. Moreover, at the time of writing, ANTLR 4 was recently announced and promises many changes and improvements, especially in terms of performance and usability. Instead of defining an own AST by using operators and rewrite rules, ANTLR 4 automatically builds a parse tree and generates interfaces to access this tree via the visitor pattern.

Moreover, the cloud profile for Microsoft Azure can be enhanced with additional information and characteristics. Further constraint validators could utilize the additional information to detect more restrictions that Azure imposes on C# systems.

References

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Commun. ACM*, 53:50–58, April 2010. ISSN 0001-0782. doi: 10.1145/1721654.1721672.

[2] Azure. Microsoft Azure Platform. `http://www.microsoft.com/windowsazure/`, last access: 2011-09-23.

[3] Muhammad Ali Babar and Muhammad Aufeef Chauhan. A tale of migration to cloud computing for sharing experiences and observations. In *Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*, SECLOUD '11, pages 50–56, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0582-2. doi: 10.1145/1985500.1985509.

[4] Jean Bovet and Terence Parr. ANTLRWorks: An ANTLR Grammar Development Environment. *Softw. Pract. Exper.*, 38(12):1305–1332, October 2008. ISSN 0038-0644. doi: 10.1002/spe.v38:12.

[5] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. MoDisco: A Generic and Extensible Framework for Model-Driven Reverse Engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 173–174, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0116-9. doi: 10.1145/1858996.1859032.

[6] E.J. Chikofsky and II Cross, J.H. Reverse Engineering and Design Recovery: A Taxonomy. *Software, IEEE*, 7(1):13 –17, jan. 1990. ISSN 0740-7459. doi: 10.1109/52.43044.

[7] S. Ducasse and D. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *Software Engineering, IEEE Transactions on*, 35(4):573 –591, july-aug. 2009. ISSN 0098-5589. doi: 10.1109/TSE.2009.19.

[8] S. Efftinge and M. Völter. oAW xText: A Framework for Textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006.

[9] EMF. Eclipse Modeling Framework (EMF). `http://www.eclipse.org/modeling/emf/?project=emf`, last access: 2011-09-22.

[10] Moritz Eysholdt and Heiko Behrens. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 307–309, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0240-1. doi: 10.1145/1869542.1869625.

[11] Charles N. Fischer, Ron K. Cytron, and Richard J. LeBlanc (Jr.). *Crafting a Compiler*. Addison-Wesley, 2010. ISBN 9780136067054.

[12] Bryan Ford. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X. doi: 10.1145/964001.964011.

[13] Sören Frey and Wilhelm Hasselbring. The CloudMIG approach: Model-based migration of software systems to cloud-optimized applications. *International Journal on Advances in Software*, 4(3 and 4), 2011. to appear.

[14] Sören Frey and Wilhelm Hasselbring. An extensible architecture for detecting violations of a cloud environment's constraints during legacy software system migration. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR 2011)*, pages 269–278. IEEE Computer Society, March 2011. ISBN 978-0-7695-4343-7. doi: 10.1109/CSMR.2011.33.

[15] Sören Frey, Wilhelm Hasselbring, and Benjamin Schnoor. Automatic conformance checking for migrating software systems to cloud infrastructures and platforms. *Journal of Software Maintenance and Evolution: Research and Practice*, 2012. ISSN 1532-0618. doi: 10.1002/smr.582.

[16] Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler. Technical report, Bell Telephone Laboratories, 1975.

[17] KDM. Knowledge Discovery Metamodel (KDM). `http://www.omg.org/spec/KDM/`, last access: 2011-09-22.

[18] A. Khajeh-Hosseini, I. Sommerville, J. Bogaerts, and P. Teregowda. Decision Support Tools for Cloud Migration in the Enterprise. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 541 –548, july 2011. doi: 10.1109/CLOUD.2011.59.

[19] Rainer Koschke. Architecture Reconstruction. In Andrea De Lucia and Filomena Ferrucci, editors, *Software Engineering*, volume 5413 of *Lecture Notes in Computer Science*, pages 140–173. Springer Berlin / Heidelberg, 2009. ISBN 978-3-540-95887-1. doi: 10.1007/978-3-540-95888-8_6.

[20] M. E. Lesk and E. Schmidt. UNIX Vol. II. chapter Lex: A Lexical Analyzer Generator, pages 375–387. W. B. Saunders Company, Philadelphia, PA, USA, 1990. ISBN 0-03-047529-5. URL `http://dl.acm.org/citation.cfm?id=107172.107193`.

[21] Scott McPeak and George Necula. Elkhound: A Fast, Practical GLR Parser Generator. In Evelyn Duesterwald, editor, *Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, pages 2725–2725. Springer Berlin / Heidelberg, 2004. ISBN 978-3-540-21297-3. doi: 10.1007/978-3-540-24723-4_6.

[22] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Special Publication 800-145, September 2011. URL `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`. last access: 2012-03-14.

[23] Bernhard Merkle. Textual modeling tools: overview and comparison of language workbenches. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 139–148, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0240-1. doi: 10.1145/1869542.1869564.

[24] L. Moonen. Generating robust parsers using island grammars. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 13 –22, 2001. doi: 10.1109/WCRE.2001.957806.

[25] .NET. Microsoft .NET Framework. `http://www.microsoft.com/net`, last access: 2011-09-23.

[26] OMG. Object Management Group (OMG). `http://www.omg.org/`, last access: 2011-09-22.

[27] Terence Parr and Kathleen Fisher. LL(*): the foundation of the ANTLR parser generator. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 425–436, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993548.

[28] Terence J. Parr and Russell W. Quong. Adding Semantic and Syntactic Predicates to LL(k): pred-LL(k). In *In Computational Complexity*, pages 263–277. Springer-Verlag, 1994.

[29] Terence J Parr, T. J. Parr, and R W Quong. ANTLR: A Predicated-LL(k) Parser Generator, 1995.

[30] B.P. Peddigari. Unified Cloud Migration Framework - Using factory based approach. In *India Conference (INDICON), 2011 Annual IEEE*, pages 1 –5, dec. 2011. doi: 10.1109/INDCON.2011.6139639.

[31] Ricardo Pérez-Castillo, Ignacio García-Rodríguez de Guzmán, and Mario Piattini. Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems. *Computer Standards & Interfaces*, 33(6):519 – 532, 2011. ISSN 0920-5489. doi: 10.1016/j.csi.2011.02.007.

[32] B.P. Rimal, Eunmi Choi, and I. Lumb. A Taxonomy and Survey of Cloud Computing Systems. In *INC, IMS and IDC, 2009. NCM '09. Fifth International Joint Conference on*, pages 44 –51, aug. 2009. doi: 10.1109/NCM.2009.218.

[33] R. S. Scowen. Extended BNF — A Generic Base Standard, 2008. URL `http://www.cl.cam.ac.uk/~mgk25/iso-ebnf.html`.

[34] Yunlin Su and Song Y. Yan. *Principles of Compilers*. Higher Education Press, Beijing and Springer-Verlag Berlin Heidelberg, 2011. ISBN 9783642208355.

[35] Van Tran, Jacky Keung, Anna Liu, and Alan Fekete. Application migration to cloud: a taxonomy of critical factors. In *Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*, SECLOUD '11, pages

22–28, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0582-2. doi: 10.1145/1985500.1985505.

[36] Unicode. Unicode 6.1.0. `http://www.unicode.org/versions/Unicode6.1.0/`, last access: 2012-03-29.

[37] S. Venugopal, S. Desikan, and K. Ganesan. Effective Migration of Enterprise Applications in Multicore Cloud. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 463 –468, dec. 2011. doi: 10.1109/UCC.2011.76.

[38] C.-Y. Yam, A. Baldwin, S. Shiu, and C. Ioannidis. Migration to Cloud as Real Option: Investment Decision under Uncertainty. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 940 –949, nov. 2011. doi: 10.1109/TrustCom.2011.130.

[39] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud Computing: State-of-the-Art and Research Challenges. *Journal of Internet Services and Applications*, 1(1): 7–18, May 2010. ISSN 1867-4828. doi: 10.1007/s13174-010-0007-6.

Glossary

ANTLR
ANother Tool for Language Recognition

API
Application Programming Interface

AST
Abstract Syntax Tree

BOM
Byte Order Marker

CEM
Cloud Environment Model

CIL
Common Intermediate Language

Cloud Computing
Cloud Computing is an approach to reduce over-provisioning of large-scale data centers and simultaneously offers a way of cost-efficient resource allocation for cloud users

CLR
Common Language Runtime

DLL
Dynamic Link Library

EBNF
Extended Backus–Naur Form

EMF
Eclipse Modeling Framework

IaaS

Infrastructure-as-a-Service

IDE

Integrated Development Environment

Infrastructure-as-a-Service

Infrastructure-as-a-Service is a cloud service model that represents the delivery of computing resources as utilities

KDM

Knowledge Discovery Meta-Model

Knowledge Discovery Meta-Model

Object Management Group's Knowledge Discovery Meta-Model represents information related to existing software assets and their operational environments. It is especially independent of the used programming language and operating system

LOC

lines of code

NIST

National Institute of Standards and Technology

Object Management Group

The *Object Management Group* is an international, open membership, not-for-profit computer industry consortium that develops enterprise integration standards

OMG

Object Management Group

OS

operating system

PaaS
: Platform-as-a-Service

Platform-as-a-Service
: Platform-as-a-Service is a cloud service model that represents the delivery of a self-maintaining platform

SaaS
: Software-as-a-Service

SMM
: Structured Metrics Meta-Model

Software-as-a-Service
: Software-as-a-Service is a cloud service model that represents the provisioning of a particular software system or application

Structured Metrics Meta-Model
: Object Management Group's Structured Metrics Meta-Model represents measurement information related to software, its operation, and its design. It allows both to define metrics and to store metric-related measurement results of arbitrary software systems

UML
: Unified Modeling Language

VM
: virtual machine

XMI
: XML Metadata Interchange

XML
: Extensible Markup Language

Appendices

A The Class `Expression`

```
1  public class Expression {
2
3    private boolean expression;
4
5    public boolean isTrue() {
6      return expression;
7    }
8    // custom methods used in lexer actions
9    public void set(boolean expression) {
10     this.expression = expression;
11   }
12   public void set(Expression exprParam) {
13     this.expression = exprParam.expression;
14   }
15   public void or(Expression expr1, Expression expr2) {
16     this.expression = expr1.expression || expr2.expression;
17   }
18   public void and(Expression expr1, Expression expr2) {
19     this.expression = expr1.expression && expr2.expression;
20   }
21   public void equal(Expression expr1, Expression expr2) {
22     this.expression = expr1.expression == expr2.expression;
23   }
24   public void unequal(Expression expr1, Expression expr2) {
25     this.expression = expr1.expression != expr2.expression;
26   }
27   public void not(Expression expr) {
28     this.expression = !expr.expression;
29   }
30 }
```

B Attachments

One CD labeled *Masterthesis attachment - Christian Wulf* containing

- the thesis as pdf-document and
- the source code of the created programs.