INSTITUT FÜR INFORMATIK

# A Three-Phase Approach to Efficiently Transform C# into KDM

Christian Wulf, Sören Frey,
and Wilhelm Hasselbring

Bericht Nr. 1211
August 2012
ISSN 2192-6247


CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

# A Three-Phase Approach to Efficiently Transform C# into KDM

Christian Wulf, Sören Frey, and Wilhelm Hasselbring
*Software Engineering Group*
*University of Kiel*
*24118 Kiel, Germany*
{*chw, sfr, wha*}*@informatik.uni-kiel.de*

***Abstract*—The Knowledge Discovery Metamodel (KDM) of the Object Management Group (OMG) is used in diverse research areas for describing software artifacts. It was recently adopted as standard ISO/IEC 19506 and its source, code, and action packages are highly suited for enabling language-independent source code analysis. However, a program needs to be transformed to KDM before corresponding source level metrics can be computed. To be of practical use, such a transformation (1) has to be resource-efficient and (2) ideally can be constructed on the basis of existing grammars to mitigate construction effort for a specific programming language.**

**In this paper, we present such an efficient transformation for C# that is structured along three fundamental phases covering distinct sub-transformations for the types, members and methods, and statements. As our approach systematically analyzes and re-engineers existing grammars and integrates appropriate decompilers, it provides insights for fluently building those program transformations in general. Our quantitative evaluation uses three C# open source systems and an industrial software from the financial sector. It shows that our approach can be successfully applied to these systems and that the transformation can efficiently transform the programs to KDM while keeping resource demand low.**

***Keywords*-Program transformation, C#, KDM, Grammar re-engineering**

## I. Introduction

With the introduction of the Knowledge Discovery Meta-Model (KDM), [1] the Object Management Group (OMG) has released a language-independent specification for representing software systems on various abstraction levels. Initially, it was developed to support software modernization scenarios. However, its source, code, and action packages are highly suited for enabling arbitrary language-independent source code analyses. KDM was recently declared as standard ISO/IEC 19506 of the International Organization for Standardization and gains traction in industry and academia [1]. For this reason, there is a need for transformations that extract KDM representations from software systems implemented in different programming languages.

Those transformations need to be resource-efficient and reasonably fast. For example, considering use cases such as recurring transformation executions due to repeated calculations of source code metrics that accompany incremental modifications. However, constructing those transformations is most often cumbersome and time-consuming. Here, the construction effort can be diminished when building upon pre-existing grammars, i.e., reusing and re-engineering grammars and also preferably existing parsers, to create an AST that can be utilized for generating KDM instances.

We present such a transformation that efficiently extracts a KDM model from C#-based software systems. We originally developed this transformation in Java for C# [2] to show the flexible applicability of the KDM-based cloud migration assistant tool CloudMIG Xpress [3] on different software systems. Our transformation builds upon a modular plug-in architecture separating the core transformation component from the specific plug-in structure for CloudMIG Xpress. In this way, the transformation can also be used as a command line tool or smoothly be integrated into other applications. For the latter use case, we provide an easy-to-use API.

Our transformation utilizes the parser generator ANother Tool for Language Recognition (ANTLR) [4] in order to generate a C# parser from a grammar specification. It is divided into three sub-transformations that first transform the types, then the member declarations and method definitions, and finally the statements of the given software system.

As our approach systematically analyzes and re-engineers existing grammars and integrates decompilers to efficiently produce a KDM representation, it provides insights for fluently building those program transformations in general.

To demonstrate the applicability and efficiency of our approach, we use three C# open source systems and a library for the assessment and risk control of financial products. The latter was provided by a large German bank in the context of our project DynaMod [5]. Our evaluation shows that the runtime and memory consumption of our transformation scales linearly with respect to a system's size.

The paper is structured as follows. In Section II, we give an overview on KDM. Section III describes the steps we took to build a C# parser. In Section IV, we present our three-phase transformation approach. Then, Section V evaluates the transformation concerning its runtime and memory consumption. Afterwards, we discuss the related work in Section VI. Finally, Section VII concludes the paper and presents future work.

[1] http://www.omg.org/spec/KDM/, last accessed: 2012-08-24

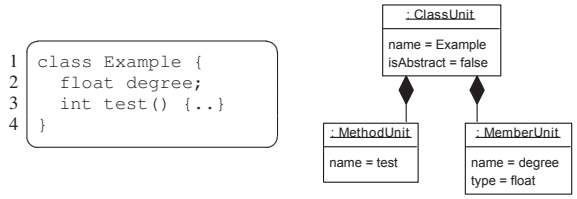



Figure 1: A C# example and a corresponding, simplified KDM instance

## II. The Knowledge Discovery Meta-Model

The Knowledge Discovery Meta-Model (KDM) is a specification for representing information related to existing software systems. The Object Management Group (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. The lowest layer, called *Infrastructure Layer*, consists of the three packages *Core*, *KDM*, and *Source*. It represents common meta-model elements for higher layers and describes the physical artifacts of the software system, e.g., source code and build files. This layer depends on no other layer.

The next overlying layer is called the *Program Elements Layer* and includes the code and action package already mentioned before. 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*.

For our transformation, we will use the *Infrastructure Layer* and the *Program Elements Layer*, especially the source, code, and action package.

Figure 1 shows an example in C# and a corresponding, simplified KDM instance that is illustrated as a UML object diagram for the sake of simplicity. Here, the C# class *Example*, method *test*, and member *degree* are mapped to corresponding instances of the KDM elements *ClassUnit*, *MethodUnit*, and *MemberUnit*, respectively.

## III. Grammar Construction via Re-Engineering

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 want to integrate the transformation into CloudMIG Xpress that is implemented in Java.

| 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 I: Available C# grammars

[a] http://antlrcsharp.codeplex.com/
[b] http://www.antlr.org/grammar/1151612545460/CSharpParser.g
[c] http://www.antlr.org/grammar/1127720913326/tkCSharp.g
[d] http://blogs.msdn.com/b/lucian/archive/2010/04/19/grammar.aspx
(a-d last accessed: 2012-08-24)

We choose the very popular and matured parser generator ANTLR that is able to generate a parser in several programming languages including Java. Thus, 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) 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 further has to consider the fact that ANTLR is an LL(*) parser and thus does not accept left-recursive rule definitions. 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 the following.

We use Google with the keywords *antlr*, *C#*, and *grammar* to search for free C# grammars in the ANTLR format. Table I lists all grammars of the first 100 results.

### A. Grammar Analysis

The first column describes what C# version is supported by the given grammar. The second column shows the required ANTLR version to compile the grammar. Column number three describes whether the author also provides an implementation for the preprocessor. 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.

We require a C# grammar that is up-to-date, i.e., it needs to conform to the current version of the C# specification. At the time of writing, this corresponds to the version 4.0. Thus, we cannot use the second and third grammar to parse and to build an Abstract Syntax Tree (AST). Moreover, a simple evaluation shows that these two grammars contain bugs and are restricted to particular use cases, for example, they do not support generics (introduced in C# version 2). Additionally, the grammar of Quentin Gregory already contains some AST nodes, most of them are not necessary for our purpose or

contain insufficient information, i.e., they are too abstract. Hence, 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 from C# to Java and translate the action code sections to equivalent Java-statements. The author uses action code for the preprocessor implementation in the lexer section, for instance. However, the preprocessor logic is not fully implemented.

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

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.

Furthermore, we found some valid C# source code that causes syntax errors when using the lexer and parser generated from this grammar. At this point, we look more closely at the delivered test framework. We add new tests consisting of the valid C# source code and execute it with the incorrect lexer and parser by means of Bradnan's test framework. All tests wrongly pass without any error message. Thus, we cannot rely on the test results anymore and from now on doubt the correctness and completeness of the whole grammar. Possibly, it would have been less error-prone if the unit test framework gUnit for ANTLR-based grammars would have been used.[2]

*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.

[2] http://www.antlr.org/wiki/display/ANTLR3/gUnit+-+Grammar+Unit+Testing, last accessed: 2012-08-24

Listing 1: Original grammar rule and modifications (bold)

```
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   ;
```

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.

### B. Lexer 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*.

In the following, we describe the development of our lexer and parser in detail. The final lexer grammar including the preprocessor as well as the parser grammar can be found as zip-file at http://www.antlr.org/grammar/list.

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 (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.

Hence, we move and remove some lexer rules. We then make some modifications for the sake of modularization and readability. If possible, we wrote programs that automate individual steps. We manually translated informal phrases in ANTLR-conform syntax. Listing 1 gives an example.

Besides having all capabilities of the EBNF standard, ANTLR grammars can additionally express some rules in a more compact and therefore more readable way. For example, we utilize ANTLR's positive closure operator + to express "one or more" occurrences of a given rule.

*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. We did not use a separate preprocessor program that returns preprocessed source code because we wanted to deliver a complete, independent Java-based parsing component.

First, we have to sort our lexer rules within the lexer grammar. For a given input, ANTLR tries to match it using the rules of the grammar in the order in which the rules are

specified. If one of the lexer rules cannot be matched because a prior rule includes it, ANTLR prints a warning message. In general, this indicates a superfluous rule or a wrong ordering. Afterwards, we move all preprocessor directives into a new grammar file that imports the lexer grammar. In this way, we separate the definition of preprocessor directives from the other lexer rules. By doing so, it is also possible to reuse the lexer rules without the preprocessor logic.

We start by instructing the lexer to skip token creation for preprocessor rules because the parser only operates on regular tokens. For this purpose, we use ANTLR's channel concept that we do not describe here.

Subsequently, we implement the necessary logic for the conditional preprocessor directives. For this purpose, we choose a stack with boolean values and integrate it into the preprocessor grammar. If a corresponding *if-directive* lexer rule matches, it should ideally evaluate a conditional expression subrule and push the return value onto the stack. In this way, the lexer can decide whether the if-body should be passed to the parser or just be skipped. Unfortunately, lexer rules cannot return values. Although parser rules can, we may also not use them 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. We first instantiate such an object whose boolean value is initialized to `false`. Then, we pass it to the conditional expression subrule. Finally, we access the potentially updated boolean value by the object's method `isTrue()` and push it onto the stack.

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's method `mTokens` that is responsible for choosing the next proper lexer rule for the current input sequence.

### C. Parser Grammar

First, we define the dependency to the preprocessor by setting the `tokenVocab` attribute in the `options` section to the name of the preprocessor 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. 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.

Finally, we still find some rules that are not correct concerning the specification. Perhaps Wischik has used an older version of the C# 4 specification or has not copied but transcribed it. In such cases, we correct those mistakes.

### D. AST Construction

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 two appropriate mechanisms of ANTLR, namely operators and rewrite rules, to define node hierarchy and to remove unwanted nodes.

## IV. The Three-Phase Approach

In this section, we describe our approach for transforming C# source code to KDM. We first give an overview and then present the transformation's three phases.

### A. Overview

Figure 2 illustrates the involved transformation parts and steps when transforming a C# system to an appropriate KDM instance.

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 in Section III. 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 Eclipse Modeling Framework (EMF)-based KDM specification to create KDM elements in Java. For resolving external libraries within the transformation process, we use either C# decompilers or a separate C# program that utilizes the .NET Reflection API. Besides integrating our component into CloudMIG Xpress, we want to enable a KDM extraction from C# for

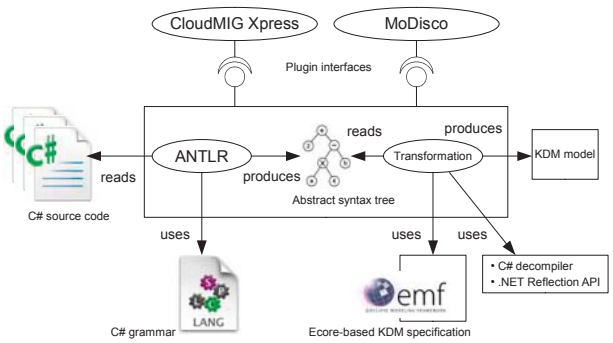



Figure 2: Transformation architecture

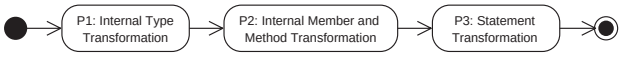



Figure 3: The three phases of our transformation

the reverse-engineering framework MoDisco [6] as well. In both cases, we implement the offered plug-in interfaces. To structure our transformation, we use the three transformation phases P1-P3 that are depicted in Figure 3. Each phase builds upon the previous one with the exception of the first phase.

### B. P1: 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. For instance, it intentionally does not transform any inheritance relations since this would require a complex and time-consuming look-up mechanism.

Considering Listing 2, our transformation would produce an XML Metadata Interchange (XMI) file with the simplified contents illustrated as UML object diagram in Figure 4.

The KDM instance with its root `Segment` element contains the inventory and code model. The latter owns a KDM `Module` and `CompilationUnit` element.

The `Module` element holds all namespaces that are newly defined by the processed source files. Since the example from Listing 2 defines the namespace `ExampleNamespace`, a corresponding KDM `Namespace` element is present in Figure 4. The *global* namespace represents the root namespace as defined in the C# specification. The `CompilationUnit` represents the logical element of the example file. In P1, it only holds a reference to the corresponding inventory model element (here indicated by the *name* attribute) and the file's type definitions. Since the example from Listing 2 defines a class `A`, the `CompilationUnit` in Figure 4 owns a corresponding KDM `ClassUnit` element.

Listing 2: A simple C# example in a file `example.cs`

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

### C. P2: Internal Member Declaration and Method Definition Transformation

The second transformation phase is responsible for transforming member declarations and method definitions but 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 needs to look up external types only.

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, its modifiers, return type, type parameters, and formal parameters are mapped to corresponding KDM elements, too.

After applying the transformation phases P1 and P2, we get an XMI file with the contents illustrated in Listing 5. Here, we again omit some details for the sake of simplicity.

Besides the KDM elements that resulted from P1 (marked in gray), we now can also find a KDM `MemberUnit` element and a KDM `MethodUnit` element, for example.

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` are located in the external code model (omitted in Listing 5 for brevity).

The `MemberUnit` element represents the member `f` of type `File`. Thus, the attribute `type` refers to the external type `System.IO.File`. Modifiers are also represented but are not shown in the example.

The `MethodUnit` element represents the method `Read`. Its `type` attribute refers to the method's child KDM element `Signature` that contains the return type and the formal

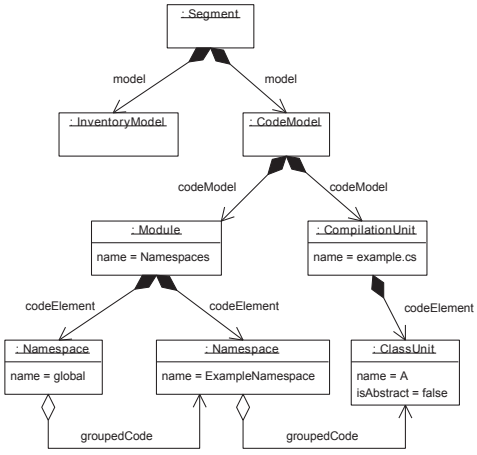



Figure 4: Simplified, visualized KDM instance generated from Listing 2 in the phase P1

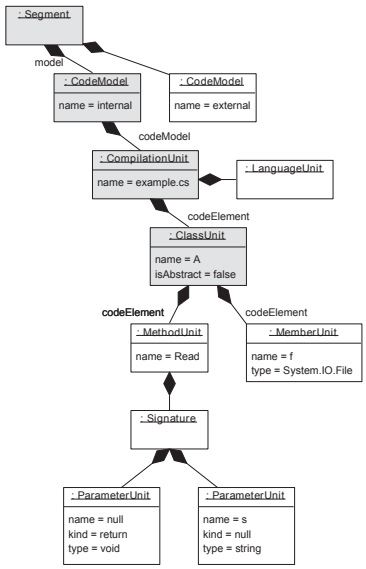



Figure 5: Simplified, visualized KDM instance generated from Listing 2 in the phases P1 and P2. Elements created in P1 are colored gray, elements from P2 are colored white.

parameters, represented both as KDM `ParameterUnits`. To differentiate between the return type and the parameters, the return type's attribute `kind` is set to `return`. Its `type` attribute refers to the KDM representation of the C# type `void` (here only illustrated as text). 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.

### D. P3: Statement Transformation

The final third transformation phase is responsible for mapping C# statements, i.e., especially member initializers and method bodies are transformed. Figure 6 demonstrates the method body of the `Read` method from Listing 2 resulting from the transformation after all three phases.

Since we want to be compatible to MoDisco's implementation of the KDM specification, we adapt its action package conventions. If we look at Figure 6, we see a KDM `BlockUnit` that represents the method body. It contains all statements as KDM `ActionElements`. `System.Console.WriteLine` is not only a method invocation, but also generally an expression statement. Thus, the *method invocation* `ActionElement` 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 `from` attribute refers to the *method invocation* `ActionElement`, its `to` attribute points to the `MethodUnit` that represents the method `System.Console.WriteLine`. As this method is extracted from the .NET framework, it is contained in the external code model.

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 transforms it including the considered method.

We provide two mechanisms to extract types from .NET libraries. The former makes use of a separate C# program that extracts the necessary information by means of the Reflection API. The program writes the information to a text file which our transformation component subsequently reads in. The latter mechanism utilizes available C# decompilers. Necessary libraries are decompiled to common C# files and parsed by our transformation component. We give a detailed evaluation of the nine most popular C# decompilers in [2].

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 V, we verify our statement by providing accuracy and performance analyses.

## V. Evaluation

This section presents our evaluation methodology and the conducted experiments.

### A. Methodology

When performing the accuracy analysis, we use the matured C# analysis tool NDepend[3] to compare the number of

[3] http://www.ndepend.com/, last accessed: 2012-08-24

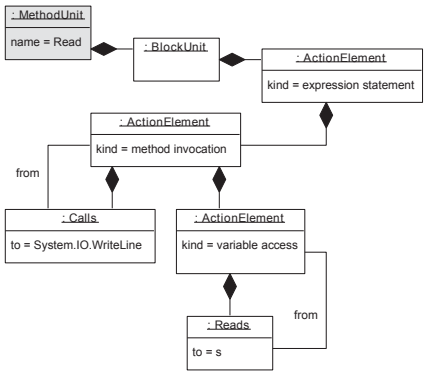



Figure 6: Simplified, visualized KDM instance generated from Listing 2 in the phases P1 to P3

namespace, type, and method definitions for assessing the completeness of our transformation.

When performing the efficiency 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. For each phase P1-P3, we additionally measure the maximal memory consumption out of four measurements per phase.

We use four applications for the efficiency evaluation. We choose two smaller, one medium, and one larger C# project to analyze the scalability with sufficient precision. Three of the applications are released under an open source license, the other one is a C#-based library for the assessment and risk control of finance products of a large German bank. Table II shows the open source applications with additional information. The accuracy analysis utilizes just the bank library, whereas the efficiency analysis employs all of the applications.

### B. Accuracy Analysis

In this section, we investigate the accuracy of our transformation component. We begin by briefly considering the correctness. Then, we check the completeness of the KDM instance that results from the transformation of the library.

*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 complex phases, we should perform an intensive correctness analysis to improve the reliability of and the trust in our transformation.

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 our work. Currently, we provide several JUnit tests that cover the transformation for type definitions, member declarations, method definitions, and a set of C# statements.

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.

| Name | Description | Dependencies | URL | Last Update |
|---|---|---|---|---|
| Sharp-Develop | C# IDE with debugger and testing environment | .NET and many more | [a] | 2012-03-04 |
| NAnt | .NET build tool | .NET and 3 more | [b] | 2011-10-22 |
| RAIL | Runtime Assembly Instrumentation Library | .NET and 2 more | [c] | 2005-01-19 |

Table II: Open source applications used for the evaluations

[a] http://sourceforge.net/projects/sharpdevelop/, last accessed: 2012-08-24
[b] http://nant.sourceforge.net/, last accessed: 2012-08-24
[c] http://rail.dei.uc.pt/index.htm, last accessed: 2012-08-24

*Completeness Analysis:* In the following, we perform a completeness analysis by means of the industrial C# library. 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.

*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 elements. In the context of this paper, 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 type relationships and statements.

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

*3) Scenarios:* We execute our transformation component on the C# library and compare the number of namespaces, type definitions (without delegates), method definitions, and member declarations (without constants) with the ones determined by NDepend.

*4) Results:* Table III shows the analysis and transformation results of NDepend (second column) and our transformation component (third 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.

*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

| Entity | NDepend | Our Transformation |
|---|---|---|
| Namespaces | 109 | 113 (+4, +3.7 %) |
| Types | 1,355 | 1,359 (+4, +0.3 %) |
| Methods | 9,327 | 6,250 (-3077, -33.0 %) |
| Members | 7,169 | 8,356 (+1187, +16.6 %) |

Table III: Completeness analysis of the library

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 `MemberUnits` and the lack of support for constant members.

*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.

### C. Efficiency Analysis

Below, we evaluate the ANTLR-generated parser and our transformation with respect to their performance and memory consumption. Section V-C1 to Section V-C6 present the goals, the experimental setting, the scenarios, the results, a discussion, and finally the threats to validity.

*1) Goals:* The following experiments evaluate the performance of the ANTLR-generated parser and our transformation component. Furthermore, the experiments analyze whether they scale according to appropriate size metrics: number of files, lines of code (LOC), and file size.

We expect that our ANTLR-generated parser 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. We also expect the processing period of our transformation to grow linear in time for larger systems.

*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, LOC, the summarized file size, and the number of C# types of the used applications. We need the information for the different

| Application | Number of Files | LOC[a] | Project Size | C# Types[b] |
|---|---|---|---|---|
| SharpDevelop | 6,399 | 618,565 | 25.44 MB | 8,518 |
| Bank library | 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 IV: Basic Information about the Used Applications

scenarios. The number of source files and the project size are given by the OS. The tool *cloc*[4] calculates the LOC for us. We use NDepend to determine the overall number of types that the individual applications define. Table IV displays the information about each application.

*3) Scenarios:* We define five scenarios (S1-S5) for the performance analysis. S1 and S2 serve as a performance and scalability indicator of our ANTLR-generated parser. The last three scenarios (S3-S5) 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 V-A.

In S1, we let our generated parser parse all C# source files of the given applications presented in Section V-A. In S2, we let the parser, while parsing, additionally create the AST that we have defined in Section III-D. 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.

*4) Results:* Figure 7 illustrates the memory consumption of each application within the three phases.

We can observe that the memory consumption of each application grows almost linearly from phase to phase. Even for the larger two projects, the maximal memory consumption in P2 and P3 stays below 520 MB and 610 MB, respectively. We aborted the P3 transformation for SharpDevelop because it has not terminated after more than 15 minutes. We discuss this behavior in Section V-C5 below.

Figure 8 shows the results of the scenarios. Each subfigure illustrates the results of all scenarios S1-S5 with respect to a particular project size metric. Thereby, we draw one differently colored curve for each scenario. The x-axis represents the corresponding metric so that for each curve the first measuring point (from left to right) refers to the one of the application *RAIL*, the second to *NAnt*, the third to the bank library, and the last to *SharpDevelop*.

According to the amount of input, best represented by the summarized file size, the ANTLR-generated parser scales very well in parsing and building the corresponding AST structures. However, the execution times of the scenarios S1-S3 are negligible compared to those of S4 and S5.

[4] http://cloc.sourceforge.net/, last accessed: 2012-08-24

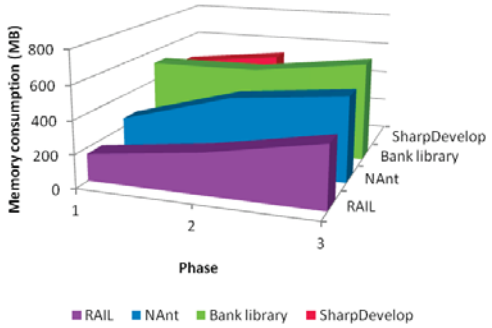



Figure 7: Memory consumption of the four applications within the three phases

Figure 8a to 8c show slightly differently growing curves for the scenario S4. For S5, however, all curves illustrate a linear or even less growth.

*5) Discussion of the Results:* The memory consumption within the phases P1 and P2 depends on the garbage collector and the logical structure of the individual applications. For example, NAnt and the bank library consumed almost the same amount of memory in P2 although NAnt is considerably smaller. We conclude that, apart from the garbage collector, NAnt comprises many members and methods because these aspects were treated in phase 2.

We only discuss the execution times of the scenarios S4 and S5 since the other ones are negligible small.

In S4, the P2 transformation of all applications took considerably more time than the P1 transformation (scenario 3) because it transforms type relationships, method definitions including their parameters, and member declarations inclusively their types. In particular, a name resolution algorithm is applied to find and correctly assign 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, the analysis shows linear curves if we exclusively consider the first three applications. Hence, we conclude that P2 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 8a to 8c only display the measurement points of *RAIL*, *NAnt*, and the *bank library*.

The resulting curve in Figure 8c indicates that even the P3 transformation scales well concerning the summarized file size. We, however, claim that the linear behavior in time depicted by Figure 8a and Figure 8b 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 Section V-C6.

*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 does currently not support all C# constructs. Thus, the final execution times of the P1-P3 transformations may increase in the future.

## VI. Related Work

In this section, we present the related work that mainly comes from the areas of grammar re-engineering and reuse and the transformation of further languages to KDM.

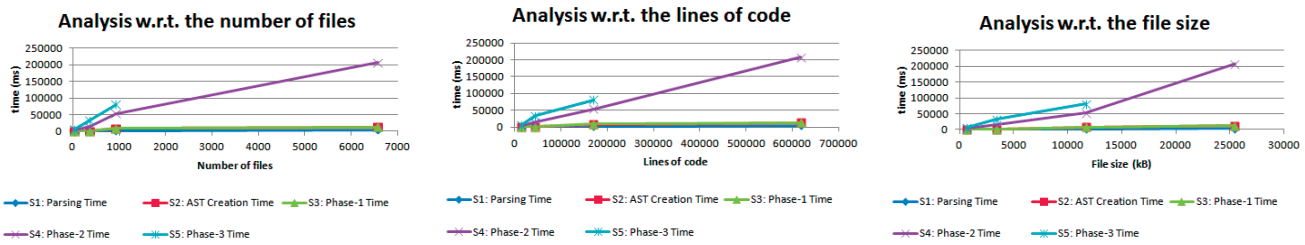
### A. Grammar re-engineering and reuse

In cases where no suitable grammar is available, it can be necessary to recover grammars from programming artifacts [7]. In contrast, we could utilize a publicly available ANTLR [4] grammar and modified it to transform C# to KDM. For scenarios that include the reuse of grammars, ANTLR provides so-called composite grammars that enable to import predefined grammar rules. Our re-engineered lexer grammar makes use of ANTLR's composite grammar functionality. Those concepts for reusing grammars are common in other parser generators as well. For example, the Xtext framework [8] for developing DSLs employs so-called grammar mixins for this purpose.

Basic strategies for re-engineering grammars are described in [9]. Considering the categorization that is proposed there, our performed adaptations result in a new *beyond equivalence* grammar and induce an *enrichment relation*. For example, to support C# preprocessor directives we had to extend the corresponding rules with semantic actions to track macro variables. Reusing grammars that incorporate such embedded semantic actions can be facilitated using the concept of prototype grammars [10]. A revision control model is employed by prototype grammars to cope with changes that incorporate arbitrary semantic actions.

### B. Transformation to KDM

The reverse-engineering framework MoDisco [6] can extract KDM models from Eclipse-based Java projects. The produced models cover KDM's source, code, and action packages. We provide a plugin that enables using our transformation component via the MoDisco UI. Here, the C# code does not need to be present as an Eclipse project.




(a) Results w.r.t. the number of files (b) Results w.r.t. the lines of code (c) Results w.r.t. the summarized file size

Figure 8: Analysis results w.r.t. the three different project size metrics

Several other tools exist that provide support for extracting KDM models. For example, the commercial software BLU-AGE ® [11] can transform Cobol code to KDM. A focus lies on KDM's source, code, and action packages as well. KDM was also used to represent SQL [12]. Corresponding extensions to KDM enable to augment KDM code models with KDM stereotypes that describe the SQL statements.

The authors in [1] list several academic and industrial projects that utilize KDM. Besides the languages mentioned before, those projects incorporate C, C++, and Ada. However, to the best of our knowledge, our evaluation is the first of its kind that publishes a detailed efficiency analysis considering a transformation from source code to KDM.

## VII. Conclusion and Future Work

We presented an approach to efficiently transform C# programs to KDM. It is composed of three fundamental phases covering distinct sub-transformations for the types, members and methods, and statements. We analyzed publicly available C# grammars and tailored the best-suited candidate. To cope with scenarios where source code artifacts are missing, we integrated support for smoothly incorporating a decompiler.

To demonstrate the applicability and efficiency of our approach, we utilized three C# open source systems and an industrial software from the financial sector. Our evaluation shows that the resource demand and the time needed to transform C#-based software to KDM scales linearly with respect to a system's size.

By parallelizing the individual phases and by reducing the number of parser passes, we plan to further increase the performance of the transformation in future work. Moreover, at the time of writing, ANTLR 4 was recently announced and it promises many improvements, especially in terms of performance and usability. We intend to exploit those improvements in a future version.

## References

[1] R. Pérez-Castillo, I. G.-R. de Guzmán, and M. Piattini, "Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems," *Computer Standards & Interfaces*, vol. 33, no. 6, pp. 519 – 532, 2011.

[2] C. Wulf, "Automatic Conformance Checking of C#-based Software Systems for Cloud Migration," Master's thesis, University of Kiel, Kiel, Germany, March 2012.

[3] S. Frey, W. Hasselbring, and B. Schnoor, "Automatic conformance checking for migrating software systems to cloud infrastructures and platforms," *Journal of Software Maintenance and Evolution: Research and Practice*, doi: 10.1002/smr.582, 2012.

[4] T. J. Parr and R. W. Quong, "ANTLR: A predicated-LL(k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[5] A. van Hoorn, S. Frey, W. Goerigk, W. Hasselbring, H. Knoche, S. Köster, H. Krause, M. Porembski, T. Stahl, M. Steinkamp, and N. Wittmüss, "Dynamod project: Dynamic analysis for model-driven software modernization," in *Proceedings of the 1st International Workshop on Model-Driven Software Migration (MDSM) 2011*, vol. 708. CEUR, 2011.

[6] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "MoDisco: A Generic and Extensible Framework for Model-Driven Reverse Engineering," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 173–174.

[7] R. Lämmel and C. Verhoef, "Semi-automatic grammar recovery," *Software: Practice and Experience*, vol. 31, no. 15, pp. 1395–1438, 2001.

[8] M. Eysholdt and H. 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*, ser. SPLASH '10. New York, NY, USA: ACM, 2010, pp. 307–309.

[9] R. Lämmel, "Grammar Adaptation," in *FME 2001: Formal Methods for Increasing Software Productivity*, ser. Lecture Notes in Computer Science, J. Oliveira and P. Zave, Eds. Springer Berlin / Heidelberg, 2001, vol. 2021, pp. 550–570.

[10] T. Parr, "The Reuse of Grammars with Embedded Semantic Actions," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, june 2008, pp. 5 –10.

[11] F. Barbier, G. Deltombe, O. Parisy, and K. Youbi, "Model Driven Reverse Engineering: Increasing Legacy Technology Independence," in *Proceedings of the Second India Workshop on Reverse Engineering*, Feb. 2011.

[12] R. Perez-Castillo, I.-R. de Guzman, M. Piattini, and O. Avila-Garcia, "On the Use of ADM to Contextualize Data on Legacy Source Code for Software Modernization," in *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, oct. 2009, pp. 128 –132.