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

Bachelor Thesis

Transforming Python into KDM to Support Cloud Conformance Checking

Written by: Alexander Clausen (`acl@informatik.uni-kiel.de`)
Born 1986-06-09 in Eckernförde

September 28, 2012

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

Hiermit versichere ich, Alexander Clausen, dass ich die Bachelorarbeit 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 Python is a well-suited language for developing web applications. This is demonstrated by the success of social networking websites like Pinterest, Reddit, Instagram, or Disqus, whose application servers mostly run Python code.

But as software ages, it often needs to go through large changes in its architecture. Doing them by hand is usually tedious and error-prone. Model-driven software development is a promising approach to this problem, as it provides automation for doing software changes. One step of this is often reverse-engineering, where a model is extracted from the existing system. The Knowledge Discovery Meta-Model (KDM) is a meta-model which can be applied here.

KDM, specified by the OMG (Object Management Group), describes a software system on different levels of abstraction in a language-independent way. It is designed as a standard for exchanging information between model-driven software modernization tools from different vendors.

KDM is used by the CloudMIG approach to support semi-automatic model-driven migration of software systems to the cloud. CloudMIG supports cloud conformance checking, which analyzes a model for violations of a set of constraints that describe the cloud environment. CloudMIG Xpress is an implementation of the CloudMIG approach.

There are already plugins to extract KDM models from Java and C# software systems. In this thesis, we present the transformation of Python software systems to KDM instances, towards the goal of supporting Python in CloudMIG Xpress. A tool for extracting a KDM model from Python is developed. As a prerequisite, a mapping between Python and KDM is defined and a multi-phase approach for transforming Python to KDM is shown.

Finally, the feasibility and performance of the approach is analyzed using three Python code bases.

Contents

List of Figures

1. Introduction

1.1. Motivation

There is a big incentive for business software developers to move their products to the cloud. IaaS and PaaS solutions free companies from having to run their own datacenter and having to worry about infrastructure details at the physical level, like maintaining hardware or getting enough energy delivered to the servers [Creeger 2009].

Also, going into the cloud, companies only pay for what they use, and can scale their application up and down depending on current usage. They no longer need to pay upfront for provisioning servers for the highest usage and find them idle most of the time.

Often, the architecture of software systems is not known in detail. Documentation is lacking, if it exists, or the system may have diverged from the state where it was documented. Also, because documentation is generally written by humans, errors may be introduced when documenting a system. Furthermore, the writer may not know what aspects are of interest for the future reader.

This is especially a problem when doing big architectural changes on large software systems. One example would be moving an existing system to the cloud. There are fundamental differences between traditional deployments and running a system in a cloud computing environment. For instance, a program running on Google App Engine for Python [Google inc.] cannot write to the filesystem, open sockets to connect to other hosts, or make syscalls. This often has the consequence that the architecture needs to be adapted, for example inserting a layer of abstraction above file management or network communication.

A possible solution to handling changes in large software systems is using a model-driven approach. By operating on models, a high degree of automation is possible and independence from programming languages can be archived.

For representing software systems, the Knowledge Discovery Meta-Model (KDM) [OMG 2011] can be used. It allows the representation of a software system on different levels of abstraction in a language-independent way.

In this thesis, we present the transformation of Python software systems to KDM instances.

1.2. Goals

In this bachelor thesis we developed a Java library to transform a Python application into a KDM instance, under the perspective of using this model for assisting a migration to a cloud environment (G1). This tool was implemented as a Java library, so it can be used in different contexts. Integration into CloudMIG Xpress is the optional goal (G2).

The transformation tool targets Python 2.7 as target platform and expects that the application source code is syntactically compatible with CPython 2.7. Python 3.x is not supported, as Google App Engine for Python does not support it yet. Also, due to its low adoption, it is unlikely to be used in a legacy application.

Applications written for alternate Python runtimes, like PyPy, Jython, or IronPython, should work, too, but there are subtle differences in semantics.[1] Also, CPython code can import C extensions, where Jython can import Java packages and IronPython can import ṄET packages. Those external packages can not be analyzed by the approach in this thesis.

To summarize, the goals are:

G1: Java library to transform Python source code into KDM

G2: Integration into CloudMIG Xpress (optional)

1.3. Document Structure

In the next section, we describe the foundations for the rest of the thesis, including a well-performing approach for another language, C#. In Section 3, We describe our approach for transforming Python to KDM, and why it is necessary to divert from the three-phase-approach. Then, in Section 4, we evaluate the feasibility of our approach by analyzing transformation runs on different code bases. Finally, in Section 6, we draw a conclusion and possible directions for future work.

[1]PyPy vs. CPython: http://doc.pypy.org/en/latest/cpython_differences.html, last visit 2012-05-10. At the time of writing, there seems to be no up-to-date documentation on the differences between CPython and IronPython or the differences between CPython and Jython

2. Foundations and Technologies

2.1. Reverse Engineering

According to [Chikofsky and Cross II 1990], reverse engineering is the process of analyzing a subject system to

- identify the system's components and their interrelationships and
- create representations of the system in another form or at a higher level of abstraction.

In the context of model-driven engineering, the resulting artifact is a model. Reverse engineering can be done by statically analyzing the source code, or by tracing the running application, for example using the Kieker monitoring framework [Rohr et al. 2008].

If the source code is not available, the process may also involve decompiling or disassembling existing application binaries or bytecode.

Forward engineering denotes the opposite direction: moving from a high-level description to the physical implementation of a system [Chikofsky and Cross II 1990].

2.2. Model Transformation

A model transformation is a program that takes a model as input, which conforms to a specific metamodel, and outputs another model that conforms to a different metamodel. There are several general approaches to model transformation, including a direct-manipulation transformation, relational, operational, graph-transformation-based, template-based, structure-driven, and hybrid [Czarnecki and Helsen 2003]. For our thesis, we chose the direct-manipulation approach, where the target model is manipulated using a general-purpose programming language (Xtend) using the Ecore API of KDM.

2.3. Software Modernization

Software modernization is the process of reengineering a legacy application, to take into account new requirements and to increase the quality attributes of a system.

The software modernization process can be understood as a "horseshoe", with the left side being extraction/reverse engineering, the right side forward-engineering, and connections between the two are transformations from the old application to the new one [Kazman et al. 1998].

Software modernization includes retargeting [Kowalczyk and Kwiecinska 2009], that is, transforming the system into a new configuration, for example, a new hardware platform, or a new operating system. This includes migration to a cloud environment.

In model-driven software modernization, the process is supported by automated tools that extract models from source code and allow for refinements of the created models. Tool support for the reengineering step may include code generation.

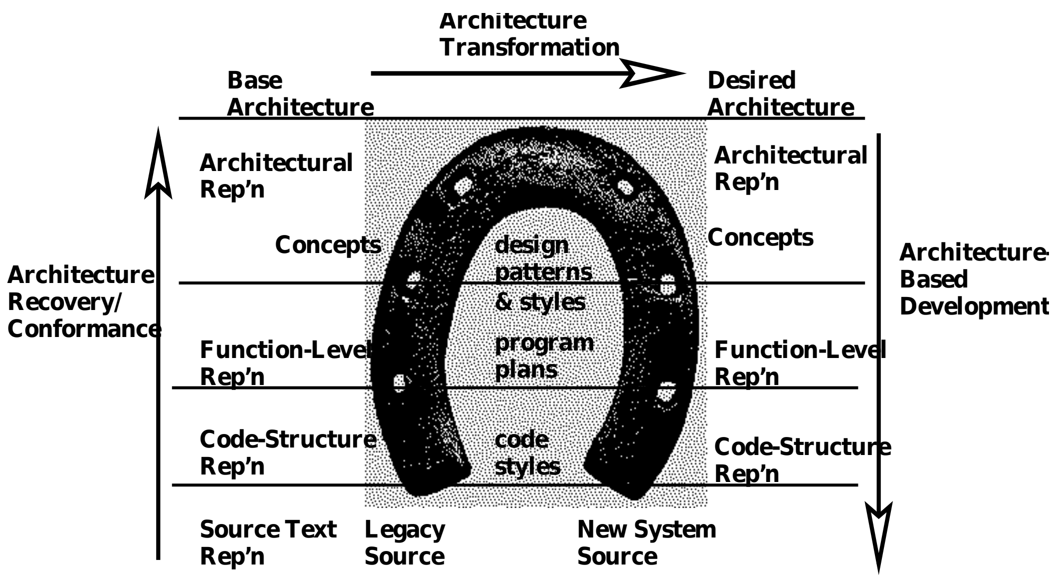


Figure 1: The Horseshoe Process [Kazman et al. 1998]

2.4. Relevant Technologies

2.4.1. KDM

The Knowledge Discovery Meta-Model (KDM), specified by the OMG [OMG 2011], describes a system's architecture on different levels of abstraction in a language-independent way. KDM is split up into layers, which are made up of individual packages. Additionaly, packages are grouped into compliance levels, that define common targets for tool interoperability.

For this thesis, the Core, kdm, Source, Code, and Action packages are used. They describe the source code files and the program elements—classes, methods, variables etc.—contained therein, but also the program behavior, like control flow and data flow.

We use existing technologies that can help with the transformation, for example EMF and Xtend. The KDM Meta-Model is available as an ECore [Eclipse Foundation] XML file, which can be loaded by EMF. Afterwards, EMF can automatically generate the Java representation of KDM and those can be used to serialize the KDM instance to an XMI file.

2.4.2. Python

Python is a dynamic, high-level programming language. Dynamic, in the context of programming languages, means that many decisions are postponed to runtime, which are done at compile-time by other languages. For example, class definitions are done at

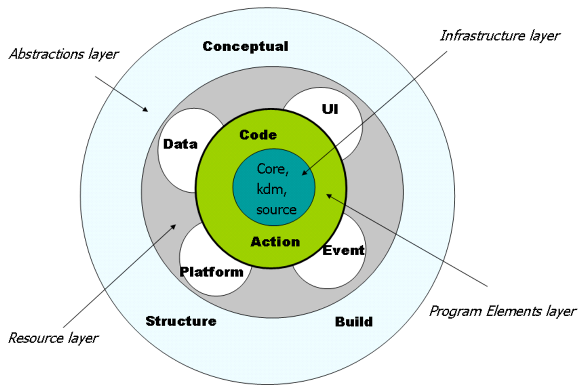


Figure 2: The 12 packages of KDM

runtime

Python is strongly, dynamically typed and features a garbage collector for memory management.

Functions, types and modules are all first-class citizens and can be passed as parameters.

There are multiple implementations; CPython being the reference implementation.

There are some challenges when trying to do static analysis on a language like Python:

- Python is a strongly, dynamically typed language, that is, variables do not have a fixed type, but values do. For an example, see Listing 1 (>>> denotes the Python prompt).

Listing 1: A Python Example

```
class Adder(object):
    def add(self, a, b):
        return a + b

def callAdd(obj, num1, num2):
    return obj.add(num1, num2)

>>> print callAdd(Adder(), 21, 21)
42
>>>
```

When doing a static analysis, the type of `obj` is generally not known without doing type inference. Anything with an method `add` which takes two arguments will be accepted at runtime. This programming style is sometimes also referred to as “duck typing”.[2] In the context of a transformation to KDM, this means that type information and call graph information will be only partially available in the resulting model. Possible solutions would be doing type inference,[3] or manual type annotations, which are only available since Python 3.x. Alternatively, such information can be added later by doing a complementary dynamic analysis. It needs to be evaluated whether it is feasible to integrate existing type inference solutions for Python into this project. Implementing complete type inference from scratch is considered out of scope for this thesis.

- The dynamic nature of Python. For example, using the `type` built-in,[4] a class can be constructed at run-time (see Listing 2).

Listing 2: Dynamically Constructing Types in Python

```
>>> Foo = type('Foo', (object,), {'getOne': lambda self: 1})
>>> f = Foo()
>>> print f.getOne()
1
>>>
```

2.4.3. Jython

Jython is an implementation of the Python language for the Java Virtual Machine. Included with Jython is an ANTLR-Grammar for parsing Python source code. Also, included with the source code of Jython is a tool called “indexer”, which provides some very useful services for us:

[2]http://docs.python.org/glossary.html#term-duck-typing, last visit 2012-05-10
[3]A good overview on type inference for Python can be found at http://lambda-the-ultimate.org/node/1519, last visited 2012-05-10
[4]Python *built-ins* are the pre-defined functions and constants that are always available

- (limited) type inference
- Python module discovery
- Parsing files using the Jython parser
- Caching the parsed ASTs

2.4.4. ATL

ATL[5] is a specialized language for doing model-to-model transforms, built on the Eclipse platform and with integration into the Eclipse IDE available. Previous work on transforming Java to KDM using ATL showed severe performance problems in the context of MoDisco [Fenner]. Even though it is unclear whether the problems stem from the language ATL itself or the rules used, we approached this problem using Xtend.

2.4.5. Xtend

As a language for doing the transformation from the Python AST into KDM, Xtend was chosen. Xtend is a statically typed programming language that compiles down to Java. It borrows a lot from Java, but adds new features, like multiple dispatch, that make it a good fit for working with models. A multi-phase approach similarly to [Wulf et al. 2012] was implemented, as it has proven to be of high performance.

2.4.6. CloudMIG and CloudMIG Xpress

A secondary goal of the thesis was integration with the CloudMIG framework[Frey and Hasselbring 2011b]. The integration was not implemented due to time constraints after consulting with the thesis advisor.

The CloudMIG framework is a semi-automatic model-driven approach to the problem of migrating software systems to the cloud. Among others, it extracts a model of the legacy application in form of a Knowledge Discovery Meta-Model (KDM, described in detail in Section 2.4.1) instance and checks for conformance with the cloud environment. The framework is independent from specific programming languages and IaaS/PaaS providers, and support for new languages can be added as plug-ins. The specific cloud environments and their constraints are described by so-called Cloud Profiles, which are instances of the Cloud Environment Model (CEM)[Frey et al. 2012]. See

CloudMIG Xpress is an implementation of the CloudMIG approach.

Currently, KDM extractors for Java [Frey and Hasselbring 2011a] and C# [Wulf 2012] are implemented. By transforming Python to KDM, a large step towards Python support for CloudMIG Xpress is done.

[5]http://www.eclipse.org/atl/, last visit 2012-05-14

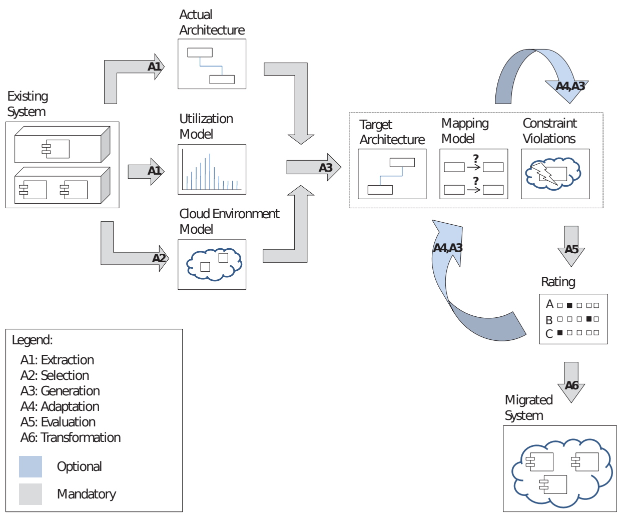


Figure 3: The CloudMIG Approach [Frey and Hasselbring 2011b]

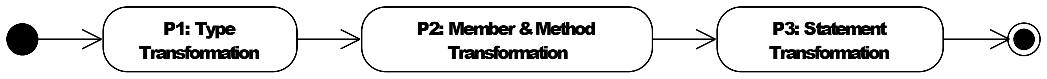

P1: Type Transformation
P2: Member & Method Transformation
P3: Statement Transformation

Figure 4: The three-phase approach for transforming C# to KDM [Wulf et al. 2012]

2.5. Three-Phase Transformation Approach

In the approach presented by [Wulf et al. 2012] for transforming C# to KDM, the transformation is done in three phases:

1. Type Transformation
2. Member & Method Transformation
3. Statement Transformation

2.5.1. Phase 1: Type Transformation

In this phase, type information and namespaces are transformed.

2.5.2. Phase 2: Member & Method Transformation

In this phase, all declarations at class-level are transformed. Types from the first phase can be referenced, for example in method signatures.

2.5.3. Phase 3: Statement Transformation

In the last phase, statements are transformed, for example member initializers and method bodies.

3. Approach

3.1. Overview

In this section, we describe our approach to the transformation of Python source code into KDM.

A tool was developed that scans an existing Python codebase, parses the source files, and transforms the resulting AST to a KDM instance.

For each file, the AST is walked by calling a dispatch method that calls the appropriate transformation method. The transformation methods then invoke the dispatch method again for relevant child AST node.

The transformation can be described as a direct-manipulation transformation, [Czarnecki and Helsen 2003] as the model is created and manipulated by using a general-purpose programming language, in this case using Xtend, through the KDM Ecore metamodel.

Our tool is built on top of the Jython indexer, which is introduced in Section 2.4.3.

As a requirement, a suitable mapping between Python language elements and KDM model elements was defined, which is detailed in Section 3.3.

3.2. The Transformation Approach

Using the Three-Phase-Approach, as detailed in 2.5, is not satisfying in combination with Python: it does not work well with the possible nestings. In Python, every statement can be nested in another compound statement (`if`, `while`, `for`, `try/except`, `with`, function and class definitions, or decorated function/class definitions). See Listing 3 for an example. This is problematic, because if we only transform class definitions in the first step, they are orphaned: their parent in the KDM model does not exist yet. We propose an adapted approach:

1. Collect import statements
2. Create a topological ordering
3. Transform toplevel code
4. Transform code below function/lambda bodies

Listing 3: Possible Statement Nesting

```
if True:
    def makeInstance():
        class SomeClass(object):
            def someMethod(self):
                import other_module
                return 42
        return SomeClass()
makeInstance()
```

3.2.1. Step 1: Collect Import Statements

This step parses each file and extracts all module dependencies, excluding those where the import is delayed, e.g. inside a function body.

3.2.2. Step 2: Create a Topological Ordering

The dependencies between the modules are represented as a directed graph, with the modules as nodes and the dependency between them as the edge. When transforming a module, all its dependencies should already be transformed, to allow references to the dependencies.

The result is a DAG that represents the necessary transform-before relationship.

3.2.3. Step 3: Transform Toplevel Code

This step transforms toplevel code, that is, code that is executed at import time. This is done in the order determined in step 2. This includes class and function definitions, but not their corresponding bodies, as they may import code from modules that are not transformed yet.

After this step, for every module, all names that are accessible from the outside are transformed into their KDM representation and can be referenced, e.g. by `Imports`.

3.2.4. Step 4: Transform Code Below Function/Lambda Bodies

In this last step, the remaining code is transformed. This includes imports inside function bodies, that can now be resolved (see Listing 3)

3.2.5. Dependencies

The transformation is run on files belonging to the software project to be transformed, but also on its dependencies, where source code is available. Step 4 is left out for dependencies, because of the associated overhead. Also, names defined inside of function bodies cannot be referenced from the outside.

Prior to running the four transformation steps, the KDM representations of basic types (`object`, `int`, `str`, and others) are transformed into KDM from static definitions.

3.3. Mapping Python Elements to KDM

To be compatible with existing tools, for example CloudMIG Xpress, the mapping from source code to KDM was kept as close to the Java KDM extraction of MoDisco as possible. A direct 1-to-1 mapping is not possible, because Python supports different language constructs than Java. For an overview over the Python language, see the Python Language Reference[6]. Often, when no corresponding language element exists in Java, and thus no reference output from MoDisco was available, the mapping was chosen similar to that of existing constructs.

As a general rule, the semantic interpretation of more dynamic features is left to the user of the model. For example, the presence of a `yield` statement in a function fundamentally changes the way the function works.

In the following sections, we give an overview of the mapping of some Python elements to KDM and point out differences to the output of MoDisco. Also, we try to point out shortcomings of the current implementation.

3.3.1. Modules and Packages

Every file with the filename suffix `.py` is a Python module. Modules can contain function/class definitions and other statements. These statements are executed when the module is imported.

The module namespace is structured by packages. Every directory that contains a file called `__init__.py` is a Python package. Python modules are then referenced, for example for importing, by their dotted module name.

Listing 4: Example Package Structure

```
toplevel/
    __init__.py
    mod1.py
    subpackage/
        __init__.py
        mod2.py
```

In Listing 4, an example package structure is given. Assuming that the `toplevel` package is in the module search path, the dotted module name of `mod2` would be `toplevel.subpackage.mod2`.

[6]http://docs.python.org/reference/index.html, last visit 2012-09-22

The namespace of the package itself, for example `toplevel.subpackage`, contains the definitions from the `__init__.py` file.

Python modules are mapped to the KDM element `Module`.

3.3.2. Literals

Numbers and strings are transformed to the KDM element `Value`. For `list` and and `tuple` literals, which do not have a Java equivalent, an `ActionElement` with `kind="list literal"` is created, with the list/tuple elements as children (`Value` if they are numbers or strings). Dictionary literals, which represent key/value mappings, are transformed to an `ActionElement` with `kind="dict"` for the container, and for each key/value pair, another `ActionElement` is created, with `kind="dict item"`. Note that a list, a tuple, or a dict can contain elements of different types.

3.3.3. Variable references

Variable references are transformed into `Reads` relations to the corresponding `StorableUnit`. If the parent expression mixes literals and variable references, the order of arguments is not preserved, for example in the context of a function call.

3.3.4. Boolean Expressions

Because of a bug in the Jython indexer, the operator in a boolean expression is not preserved in the AST, and such expressions cannot be properly transformed.

3.3.5. Classes

Python classes are transformed to KDM `ClassUnit` instances. Because of the dependency-sorted approach, all base classes[7] are guaranteed to already be transformed and the `Extends`-relationship can be created together with the `ClassUnit` instance.

Python allows arbitrary expression as base classes; see Listing 5 for an example. We only transform the case of one or more direct references of class names, like class A in the example.

Listing 5: Python Inheritance Examples

```python
class A(object):
    pass
class B(A if True else object):
    pass
class C(someFactoryFunction(), A, B):
    pass
```

[7]Python supports multiple inheritance

3.3.6. Function Definitions, Function and Method Calls

In the AST, functions and methods cannot be distinguished from each other. Likewise, function calls, method invocations and class instance creations have the same representation in the AST. So the type and context of the called object is used to determine the KDM representation.

If the parent of a function is a class, it is transformed as a `MethodUnit`, otherwise a `CallableUnit` (which is not present in Java).

In a call, if the type of object that is called is a class, the call is transformed like an instance creation, otherwise a method invocation or function call. Class instance creations and method calls are both transformed like in the Java transformation, for function calls, the `kind` and `name` attributes on the resulting `ActionElement` are set to `"function call"` instead of `"method invocation"` and the `Calls` relation points to the `CallableUnit`

Arguments of function definitions are transformed like in the Java transformation, with the addition of variadic arguments. Python has support for two different kinds of variadic arguments for functions and methods: one that captures excess positional arguments, and another that captures excess named arguments (called "keyword arguments" in Python). Both are transformed into `ParameterUnits` with `kind="variadic"`.

3.3.7. Assignments

In Python, variables are not declared as in some other languages. Names are bound (or re-bound) by assignment. There are no explicit type annotations. The left-hand side of an assignment either be a single identifier, a target list, an attribute reference, or a slicing.[8]

When assigning to a single identifier, the behaviour depends on the `global` statement. If a global statement for the identifier exists in the current block, the name is bound in the global namespace (i.e. the module). Otherwise, it is bound in the local namespace. Local namespaces are created by modules, function bodies, and class definitions.[9]

Assignments are transformed in two steps:

- Create an appropriate `Storable` KDM element, if none exists for the name
- The actual assignment, as a `ActionElement` with `kind=assignment`

For Augmented Assignments, the appropriate operator is set as `name` of the `ActionElement`.

[8]see http://docs.python.org/reference/simple_stmts.html#assignment-statements for details, last visited 2012-09-26

[9]See http://docs.python.org/reference/executionmodel.html, last visited 2012-09-26

To keep track of the `StorableUnits`, a table (called `ScopeDict`) is kept in memory for each module, class and function that maps name bindings to `StorableUnit` instances. This table also contains a pointer to its parent scope and keeps track of subscopes. A subscope is the `ScopeDict` of an object that can handle attribute access (e.g. module objects, classes, and class instances).

When the left-hand side is an attribute reference, Python evaluates the primary[10] of the left-hand side. The resulting object is then asked to perform the assignment.

In the transformation, the primary is resolved by recursively walking the subscopes of the `ScopeDict` corresponding to the local namespace. After that, it is handled like an assignment to a single identifier.

Listing 6: Attribute Assignment

```
class A(object):
    x = 7

    class Sub(object):
        b = 42

a = A()
a.x = 14
a.Sub.b = 21
```

In the expression `a.x`, `a` is the primary, and `x` is the identifier. The primary is resolved by looking in the local `ScopeDict`, which in this case corresponds to the `ScopeDict` of the module. So the `ScopeDict` of `a` is that of the class `A`. Then the identifier `x` is looked up in this `ScopeDict`, and the corresponding `StorableUnit` is found.

Similarly, the primary of `a.Sub.b` is `a.Sub`, which itself has the primary `a`.

Python also supports slicings, subscriptions and target lists on the left-hand side of an assignments; the transformation of these is left as future work.

The same expressions that are allowed on the left-hand side of an assignment are also allowed in the target list of some other compound statements, like `for`, `try`, etc. So Listing 7 is valid Python:

Listing 7: Attribute as Assignment Destination in a for-Loop

```
class A(object):
    pass
for A.varname in [1, 2, 3]:
    print A.varname
```

3.3.8. Name Resolution

In non-assignment contexts, names are resolved just like the left-hand side of the assignment, with one exception: the scope of names bound in class definitions does not extend

[10]http://docs.python.org/reference/expressions.html#primaries, last visited 2012-09-25

to functions below the class definition. See Listing 8 for an example.

Listing 8: Limited Scope of Names Bound in Class Definitions

```
class A(object):
    clsVar = 42

    print clsVar            # works

    def method(self):
        print self.clsVar   # works
        print clsVar        # NameError, because clsVar is
                            # only valid in the class scope
```

Table 1: The Selected Case Studies

Project	LOC (Project)	LOC (Dependencies)	LOC (Total)
P1	251	0	251
P2: FlussInfo	35909	394842	430751
P3: zamboni	94752	558731	653483

4. Evaluation

4.1. Overview

In this section, we evaluate the run time and memory usage of the transformation on three code bases. Also, the feasibility of our approach is analyzed by comparing the dependency graph and the names of methods/functions to the output of two existing static analysis programs.

4.1.1. Case Studies

In Table 1, we list the selected case studies that were used for the evaluation of our transformation approach, together with their size, measured in lines of code (LOC), and the size of their dependencies. The numbers were obtained using the command line tool cloc.[11] As described in Section 3.2.5, the transformation also runs on the dependencies, at least partially, so their size is relevant to the execution time and memory usage. Another dependency that is common to all three case studies is the Python Standard Library, which is not included in the LOC numbers in the table.

The project P1 is a number of small Python files, specifically written for this thesis. They do not perform a specific functionality; they were written only to test the transformation on common and more unusual language constructs. See Listings 9 and 10 for one of these test files.

[11] http://cloc.sourceforge.net/, last visited 2012-09-25

Listing 9: Example Python File From P1

```
# imports from the stdlib
from decimal import Decimal
import xml.etree

import xml.etree as etree2, xml.parsers as parsers

# create decimal object
d = Decimal("42.0")

# and print its string representation
print str(d)


# local import
def localImport():
    import simple
    import simple as alias


# strange import
class ClassLevelImport(object):
    import simple as strange_import


# conditional module-level import
if True:
    import simple as alias2


# guarded import
try:
    import xml.etree as etree3
except ImportError:
    simplejson = None


# circular local import:
def circular():
    import imports1
```

Listing 10: Example File From P1 Demonstrating Possible Nestings

```
def funWithInnerClass():
    class FunInnerClass(object):
        def add(self, a, b):
            return a + b
    return FunInnerClass


class ClassWithInner(object):
    class InnerClass(object):
        def methodOfInner(self):
            pass

    def methodOfTopLevel(self):
        class ClassInMethod(object):
            pass


def decorator(fn):
    def _wrapper(*args, **kwargs):
        return fn(*args, **kwargs)
    return _wrapper
```

The project P2, FlussInfo[12], is a proprietary web application that is a real candidate for a migration to the cloud. See Figure 4.1.1 for a screenshot of the front-end. The code base includes an online shop, a content management system and an online mapping solution, among others.

Finally, the project P3, zamboni, is an open-source project that is the code base of *Add-ons for Firefox*,[13] see Figure 4.1.1 for a screenshot.

4.2. Feasibility Analysis

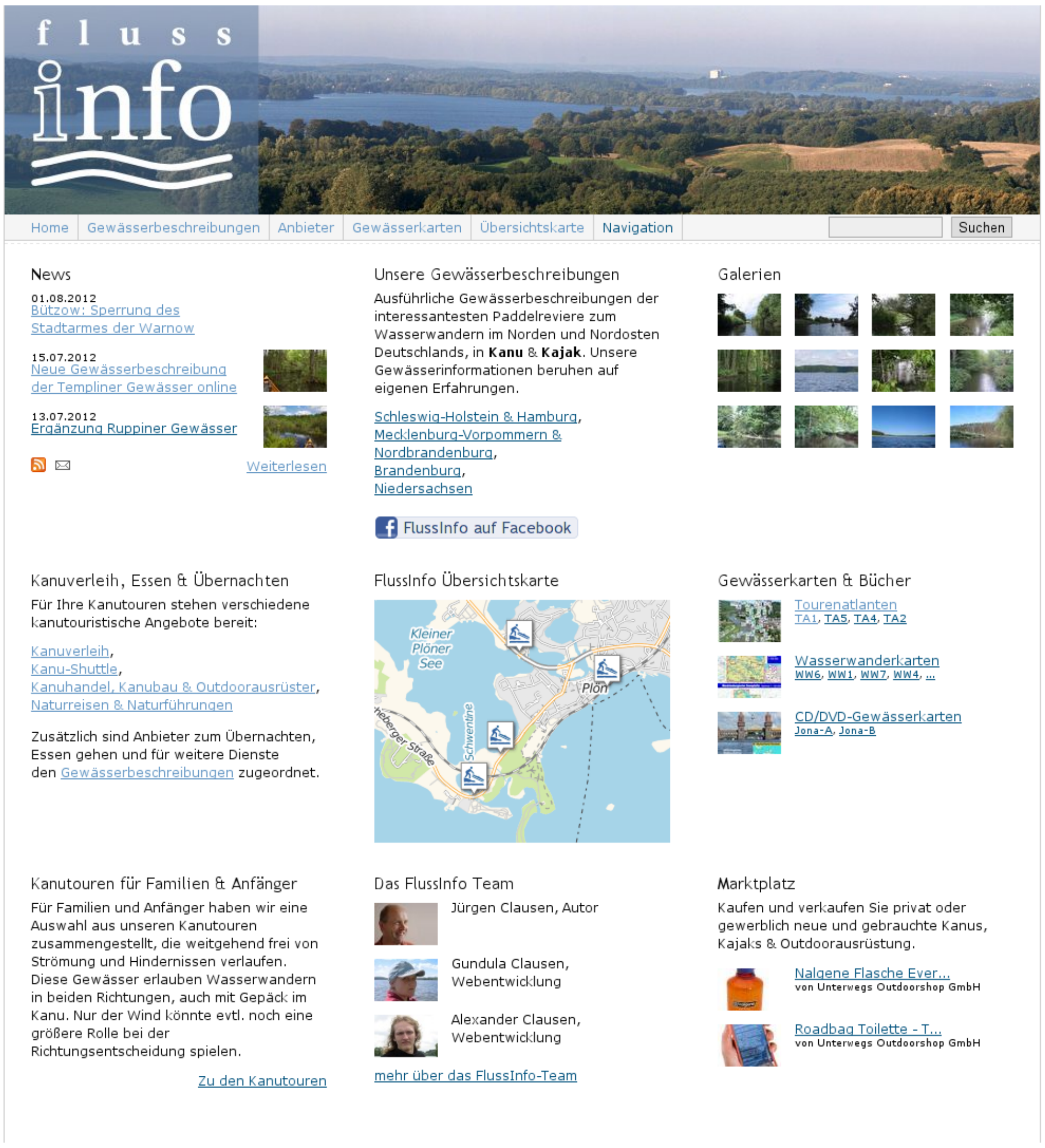
4.2.1. Goals

We want to investigate how "complete" our approach is. Are all elements from the existing software systems transformed into the KDM model?

4.2.2. Methodology

The method names in the model were compared to the output of pymetrics,[14], which creates a list of methods as a by-product of computing their McCabe metric. This list

[12]http://www.flussinfo.net/, last visit 2012-09-24
[13]https://addons.mozilla.org/, last visit 2012-05-10
[14]http://pymetrics.sf.net/, last visit 2012-09-16



News

01.08.2012
Bützow: Sperrung des Stadtarmes der Warnow

15.07.2012
Neue Gewässerbeschreibung der Templiner Gewässer online

13.07.2012
Ergänzung Ruppiner Gewässer

Weiterlesen

Unsere Gewässerbeschreibungen

Ausführliche Gewässerbeschreibungen der interessantesten Paddelreviere zum Wasserwandern im Norden und Nordosten Deutschlands, in **Kanu** & **Kajak**. Unsere Gewässerinformationen beruhen auf eigenen Erfahrungen.

Schleswig-Holstein & Hamburg, Mecklenburg-Vorpommern & Nordbrandenburg, Brandenburg, Niedersachsen

FlussInfo auf Facebook

Galerien

Kanuverleih, Essen & Übernachten

Für Ihre Kanutouren stehen verschiedene kanutouristische Angebote bereit:

Kanuverleih, Kanu-Shuttle, Kanuhandel, Kanubau & Outdoorausrüster, Naturreisen & Naturführungen

Zusätzlich sind Anbieter zum Übernachten, Essen gehen und für weitere Dienste den Gewässerbeschreibungen zugeordnet.

FlussInfo Übersichtskarte

Gewässerkarten & Bücher

Tourenatlanten
TA1, TA5, TA4, TA2

Wasserwanderkarten
WW6, WW1, WW7, WW4, ...

CD/DVD-Gewässerkarten
Jona-A, Jona-B

Kanutouren für Familien & Anfänger

Für Familien und Anfänger haben wir eine Auswahl aus unseren Kanutouren zusammengestellt, die weitgehend frei von Strömung und Hindernissen verlaufen. Diese Gewässer erlauben Wasserwandern in beiden Richtungen, auch mit Gepäck im Kanu. Nur der Wind könnte evtl. noch eine größere Rolle bei der Richtungsentscheidung spielen.

Zu den Kanutouren

Das FlussInfo Team

Jürgen Clausen, Autor

Gundula Clausen, Webentwicklung

Alexander Clausen, Webentwicklung

mehr über das FlussInfo-Team

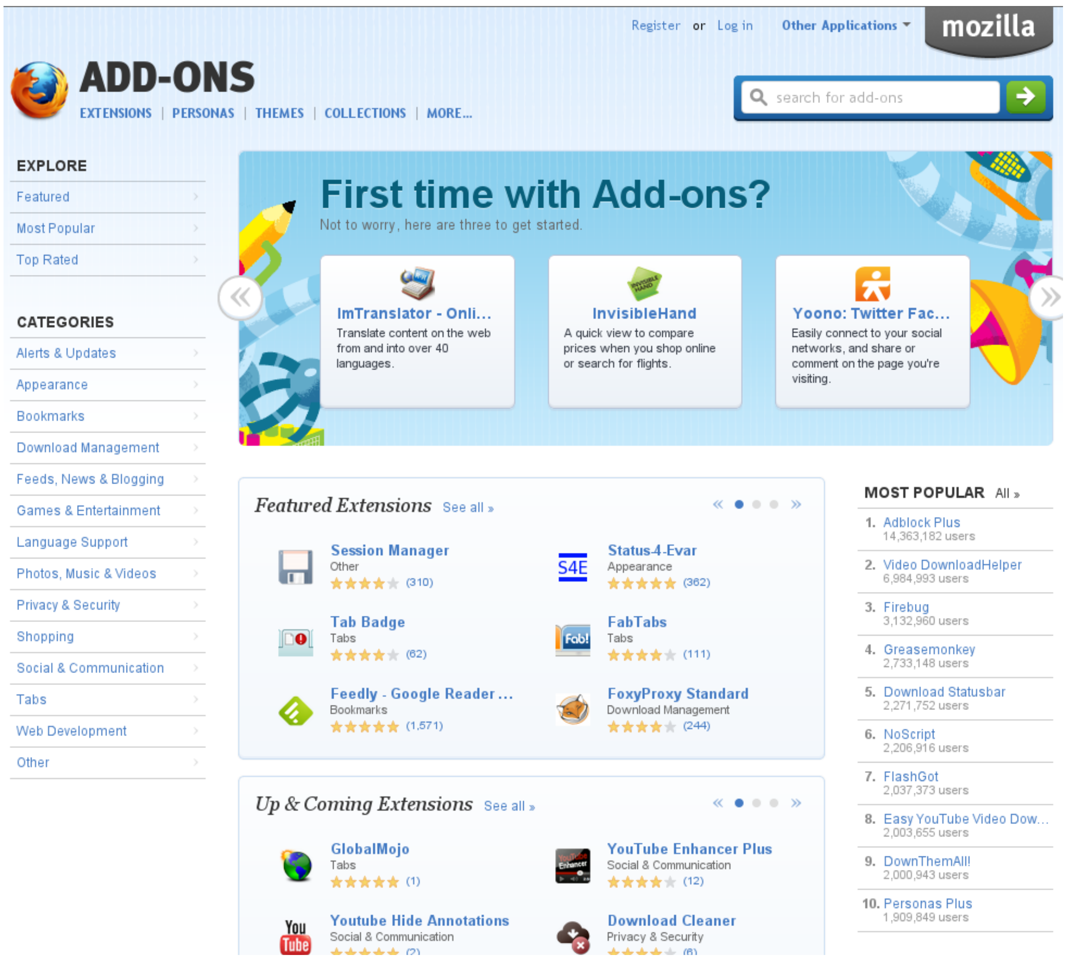
Marktplatz

Kaufen und verkaufen Sie privat oder gewerblich neue und gebrauchte Kanus, Kajaks & Outdoorausrüstung.

Nalgene Flasche Ever...
von Unterwegs Outdoorshop GmbH

Roadbag Toilette - T...
von Unterwegs Outdoorshop GmbH

Figure 5: Screenshot of FlussInfo.net, taken on 2012-09-27



Register or Log in Other Applications mozilla

ADD-ONS

EXTENSIONS | PERSONAS | THEMES | COLLECTIONS | MORE...

search for add-ons

EXPLORE
- Featured
- Most Popular
- Top Rated

CATEGORIES
- Alerts & Updates
- Appearance
- Bookmarks
- Download Management
- Feeds, News & Blogging
- Games & Entertainment
- Language Support
- Photos, Music & Videos
- Privacy & Security
- Shopping
- Social & Communication
- Tabs
- Web Development
- Other

First time with Add-ons?
Not to worry, here are three to get started.

ImTranslator - Onli...
Translate content on the web from and into over 40 languages.

InvisibleHand
A quick view to compare prices when you shop online or search for flights.

Yoono: Twitter Fac...
Easily connect to your social networks, and share or comment on the page you're visiting.

Featured Extensions See all »

Session Manager Other (310)	Status-4-Evar Appearance (362)
Tab Badge Tabs (62)	FabTabs Tabs (111)
Feedly - Google Reader ... Bookmarks (1,571)	FoxyProxy Standard Download Management (244)

Up & Coming Extensions See all »

GlobalMojo Tabs (1)	YouTube Enhancer Plus Social & Communication (12)
Youtube Hide Annotations Social & Communication	Download Cleaner Privacy & Security

MOST POPULAR All »

1. Adblock Plus
14,363,182 users
2. Video DownloadHelper
6,984,993 users
3. Firebug
3,132,960 users
4. Greasemonkey
2,733,148 users
5. Download Statusbar
2,271,752 users
6. NoScript
2,206,916 users
7. FlashGot
2,037,373 users
8. Easy YouTube Video Dow...
2,003,655 users
9. DownThemAll!
2,000,943 users
10. Personas Plus
1,909,849 users

Figure 6: Screenshot of addons.mozilla.org, taken on 2012-09-27

Table 2: Feasibility Analysis: Results

Project	Imports Missing	Functions Missing	#Imports	#Functions
P1	0 (0%)	0 (0%)	14	31
P2: FlussInfo	627 (44%)	13 (1%)	1425	1298
P3: zamboni	1819 (30%)	54 (0.5%)	5978	9881

is only present in the SQL output.

Another point of reference is the output of snakefood,[15] which is a set of Python tools that generate dependency graphs from Python files. This output is compared to an extract of `Imports` from the KDM model.

For P1, additionally, a manual comparison between the KDM result and the source files is conducted.

4.2.3. Results

Table 2 shows the results of the feasibility analysis, with the number of imports and functions that are present in the reference output from the static analysis tools but not in the KDM model.

Unfortunately, when running sfood on the FlussInfo project, it consistently picked up a different installation of the project, without giving an option to correct the paths. The resulting path data had to be corrected using an in-place search & replace operation.

A similar problem was found in the data from pymetrics, which noted an absolute import path, where, in practice, a relative import was used, which was correctly put into the KDM model. This was also corrected using an in-place search & replace operation.

The manual comparison uncovered some defects in the implementation of the transformation:

- Attribute access to the explicit `self` parameter is not transformed properly
- Unlike the Java transformation, constructors that are not present in the source file are also not present in the KDM model, and not referenced when instantiating classes.
- Empty function bodies, that only contain the `pass` statement, return None, which is not reflected in the model
- The loop variable in the `foreach`-construct is not created as a `StorableUnit`
- Empty `else`-branches of `while`-loops are not ommitted

[15] http://furius.ca/snakefood/, last visit 2012-09-25

- The built-in function `str` is missing
- Access to __name__ is not transformed properly

4.2.4. Threats to Validity

There is a bug in the Jython indexer that causes errors when processing files that use a Python feature called *explicit relative imports*. This might be responsible for some of the missing dependencies.

There might be a similar problem in the dependency analysis of zamboni as was found for FlussInfo, but the author is not as acquainted with the code base of zamboni.

4.3. Performance Analysis

4.3.1. Goals

The goal is to find out how well the execution time and memory usage scales with the project size.

4.3.2. Methodology

The evaluation was run in the following environment:

- CPU: Intel(R) Core(TM) i5-2410M CPU @ 2.30GHz
- RAM: 8GB
- Operating System: Linux 3.2.0 x86_64
- Java version: OpenJDK Runtime Environment (IcedTea6 1.11.4), OpenJDK 64-Bit Server VM (build 20.0-b12, mixed mode)

Measurement of the execution time was done using `System.currentTimeMillis()` in the transformation code, so Java VM startup time is not included. As a lot of time is spent loading the code into the Jython indexer and saving the model, we also measure a second timespan, that excludes these two and only includes the transformation from the AST to the KDM model.

Memory usage was analyzed by the `/usr/bin/time` command line tool, taking the `maxresident` measure from the output. This means that the maximum memory usage is reported here.

For transforming FlussInfo and zamboni, the available memory for the Java VM had to be adjusted to prevent running our of heap space.

For time and memory usage, we run the transformation ten times and only use the last nine; the first run is used to warm the disk cache of the operating system and the

Table 3: Performance Analysis: Results

Project	Total Execution Time	Only Transformation	Memory Usage
P1	4.8s	0.7s	282MB
P2: FlussInfo	47.5s	4.9s	3039MB
P3: zamboni	66.7s	9.2s	3243MB

on-disk AST cache of the Jython indexer. Of the resulting nine runs, we take the median value.

4.3.3. Results

The results of the performance analysis are displayed in Table 3. The column "Only Transformation" refers to the time spent traversing the AST and building the KDM model, essentially what is referred to in Section 3.2. The rest of the time is spend in the Jython indexer and saving the resulting KDM model.

The unusually high memory usage for transforming P1, which only amounts to 215 lines of code, may occur because the Python Standard Library is loaded by indexer for all Python programs.

5. Related Work

In his Master's Thesis [Wulf 2012], Wulf presents the transformation of C# to KDM, which inspired our multi-phase approach.

The project Gra2MoL (Grammar to Model Language) [Cánovas Izquierdo and García Molina 2012] develops a transformation language designed for transforming from a grammar to a metamodel. The examples include transformations from Delphi to ASTM and Java to KDM.

6. Conclusion

6.1. Summary

A transformation from Python source code to KDM was presented, which uses existing parsing infrastructure from Jython. A similar approach to the Three-Phase-Approach that was used for transforming C# to KDM was developed to fit the requirements of transforming Python code. Furthermore, the feasibility and performance of the approach was analysed using three Python code bases.

6.2. Future Work

6.2.1. Improvements of the Transformation

The feasibility analysis has shown that many small details are not covered by the transformation, and that errors in the used technology cause some modules to fail to load completely. Also, some language constructs are simply not supported yet.

Additionally, a different approach for ordering the module transformations could be looked into, that works more like the Python importer.[16]

6.2.2. Complementary Dynamic Extraction

A complementary dynamic extraction could be used to identify the types of variables where the current type inference approach fails.

6.2.3. ASTM as Intermediate Model

The OMG also specified the Abstract Syntax Tree Metamodel (ASTM)[17], which is a low-level description of the abstract syntax tree of a program. It allows a 1-to-1 mapping between code and model. Also, it can be used together with KDM, and the KDM model can be derived from the ASTM instance.

6.2.4. Union Types

Currently, the transformation assumes only one type per variable. This is not generally true, because of Python's dynamic type system. A KDM representation of union types needs to be devised

[16]See for example http://effbot.org/zone/import-confusion.htm, last visited 2012-09-25
[17]http://www.omg.org/spec/ASTM/1.0/, last visited 2012-09-25

References

[Chikofsky and Cross II 1990] E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy. pages 13–17, 1990.

[Creeger 2009] M. Creeger. Cto roundtable: Cloud computing. *Commun. ACM*, 52(8): 50–56, Aug. 2009. ISSN 0001-0782. doi: 10.1145/1536616.1536633.

[Czarnecki and Helsen 2003] K. Czarnecki and S. Helsen. Classification of model transformation approaches. 2003.

[Cánovas Izquierdo and García Molina 2012] J. Cánovas Izquierdo and J. García Molina. Extracting models from source code in software modernization. *Software & Systems Modeling*, pages 1–22, 2012. ISSN 1619-1366. doi: 10.1007/s10270-012-0270-z. URL `http://dx.doi.org/10.1007/s10270-012-0270-z`.

[Eclipse Foundation] Eclipse Foundation. Eclipse modeling framework project. URL `http://www.eclipse.org/modeling/emf/`. last visit 2012-05-06.

[Fenner] S. Fenner. Migration of software systems to platform as a service based cloud environments. Diploma Thesis, University of Kiel, Germany.

[Frey and Hasselbring 2011a] S. Frey and W. 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, Mar. 2011a. ISBN 978-0-7695-4343-7. doi: 10.1109/CSMR.2011.33.

[Frey and Hasselbring 2011b] S. Frey and W. Hasselbring. The cloudmig approach: Model-based migration of software systems to cloud-optimized applications. *International Journal on Advances in Software*, 4(3 and 4):342–353, 2011b. ISSN 1942-2628.

[Frey et al. 2012] 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*, 2012. ISSN 1532-0618. doi: 10.1002/smr.582.

[Google inc.] Google inc. Python runtime environment. URL `https://developers.google.com/appengine/docs/python/runtime`. last visit 2012-05-06.

[Kazman et al. 1998] R. Kazman, S. G. Woods, and S. J. Carrière. Requirements for integrating software architecture and reengineering models: Corum ii. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, WCRE '98, pages 154–, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8967-6.

[Kowalczyk and Kwiecinska 2009] K. Kowalczyk and A. Kwiecinska. Model-driven software modernization, 2009. Master thesis, School of Engineering, Blekinge Institute of Technology, Sweden.

[OMG 2011] OMG. Knowledge discovery meta-model version 1.3, 2011. URL `http://www.omg.org/spec/KDM/1.3/index.htm`. last visit 2012-05-06.

[Rohr et al. 2008] M. Rohr, A. van Hoorn, J. Matevska, N. Sommer, L. Stoever, S. Giesecke, and W. Hasselbring. Kieker: continuous monitoring and on demand visualization of java software behavior. In *Proceedings of the IASTED International Conference on Software Engineering*, SE '08, pages 80–85, Anaheim, CA, USA, 2008. ACTA Press. ISBN 978-0-88986-716-1.

[Wulf 2012] C. Wulf. Automatic conformance checking of c#-based software systems for cloud migration, 2012. Master thesis, Univesity of Kiel, Germany.

[Wulf et al. 2012] C. Wulf, S. Frey, and W. Hasselbring. A Three-Phase Approach to Efficiently Transform C# into KDM. Technical Report TR-1211, Department of Computer Science, Kiel University, Germany, Aug. 2012. URL `http://www.informatik.uni-kiel.de/uploads/tx_publication/TR-1211.pdf`.

A. Attachments

Attached is a CD-ROM with the source code of the transformation and the programs/scripts used for the evaluation. It also contains this document as a PDF file.