Dedicated Model Transformation Languages vs. General-purpose Languages: A Historical Perspective on ATL vs. Java

Stefan Götz[1][a], Matthias Tichy[1][b] and Timo Kehrer[2][c]

[1]*Institute of Software Engineering and Programming languages, Ulm University, James-Franck Ring, Ulm, Germany*

[2]*Department of Computer Science, Humboldt - University of Berlin, Berlin, Germany*

Keywords: Model Transformation Languages, ATL, Java, Complexity.

Abstract: Model transformations are among the key concepts of model-driven engineering (MDE), and dedicated model transformation languages (MTLs) emerged with the popularity of the MDE paradigm about 15 to 20 years ago. MTLs claim to increase the ease of development of model transformations by abstracting from recurring transformation aspects and hiding complex semantics behind a simple yet intuitive syntax. Nonetheless, MTLs are rarely adopted in practice, there is still no empirical evidence for this claim, and the argument of abstraction deserves a fresh look in the light of modern general-purpose languages (GPLs) which have undergone a significant evolution in the last two decades. In this paper, we report about a study in which we compare the complexity of model transformations written in three different languages, namely (i) the Atlas Transformation Language (ATL), (ii) Java SE5, and (iii) Java SE14; the Java transformations are derived from an ATL specification using a translation schema we developed in terms of our study. In a nutshell, we found that some of the new features in Java SE14 compared to Java SE5 help to significantly reduce the complexity of transformations written in Java. At the same time, however, the relative amount of complexity that stems from aspects that ATL can hide from the developer stays about the same. Based on these results, we indicate potential avenues for future research on the comparison of MTLs and GPLs in a model transformation context.

1 INTRODUCTION

Model transformations are among the key concepts of the model-driven engineering (MDE) paradigm (Sendall and Kozaczynski, 2003). They are a particular kind of software which needs to be developed along with an MDE tool chain or development environment. With the aim of supporting the development of model transformations, dedicated model transformation languages (MTLs) have been proposed and implemented shortly after the MDE paradigm gained a foothold in software engineering. In the literature, many advantages are ascribed to model transformation languages, such as better analysability, comprehensibility or expressiveness (Götz et al., 2020). Moreover, model transformation languages aim at abstracting from certain recurring aspects of a model transformation, claiming to hide complex semantics behind a simple yet intuitive syntax (Jouault et al., 2008; Krikava et al., 2014; Sendall and Kozaczynski, 2003; Gray and Karsai, 2003).

Nowadays, however, such claims have two main flaws. First, as discussed by Götz et al., there is a lack of actual evidence to have confidence in their genuineness (Götz et al., 2020). Second, we argue that most of these claims emerged together with the first MTLs around 15 years ago. The Atlas Transformation Language (ATL) (Jouault et al., 2006), for example, was first introduced in 2006, at a time when third generation general-purpose languages (GPLs) were still in their infancy. Arguably, these flaws are underpinned by the observation that MTLs have been rarely adopted in practical MDE (Burgueno et al., 2019).

Within our research group as well as in conversations with other researchers, the presumption that transformations can just as well be written in a GPL such as Java has been discussed frequently. In fact, in our own research, we have implemented various model transformations using a GPL; examples of this include the meta-tooling facilities of established research tools like SiLift (Kehrer et al., 2012) and

[a] https://orcid.org/0000-0001-7028-131X
[b] https://orcid.org/0000-0002-9067-3748
[c] https://orcid.org/0000-0002-2582-5557

Götz, S., Tichy, M. and Kehrer, T.
Dedicated Model Transformation Languages vs. General-purpose Languages: A Historical Perspective on ATL vs. Java.
DOI: 10.5220/0010340801220135
In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2021)*, pages 122-135
ISBN: 978-989-758-487-9

SERGe (Kehrer et al., 2016; Rindt et al., 2014), or the implementation of model refactorings and model mutations in experimental setups of more recent empirical evaluations (Schultheiß et al., 2020a; Schultheiß et al., 2020b). The presumption that model transformations can just as well be written in a GPL has been confirmed by a community discussion on the future of model transformation languages (Burgueno et al., 2019), and, at least partially, by an empirical study conducted by Hebig et al. (Hebig et al., 2018). Our argumentation for specifying model transformations using a modern GPL is mainly rooted in the idea that new language features allow developers to heavily reduce the boilerplate code that MTLs claim to abstract away from.

To validate and better understand this argumentation, we elected to compare ATL, one of the most widely known MTLs, with Java, a widespread GPL. More specifically, we compare ATL with Java in its current iteration (Java SE14) as well as at the level of 2006 (Java SE5) when ATL was introduced[1] To see how much transformation code written in Java SE5 can be improved using newer language features released in Java SE14, we also decided to compare transformations written in these two versions. Based on these goals, we devised three research questions to focus our research on:

- **RQ1:** *How much less complex can transformations be written in Java SE14 compared to Java SE5?*
- **RQ2:** *How is the complexity of transformations written in Java SE5 distributed over the different aspects of the transformation process compared to ATL?*
- **RQ3:** *How is the complexity of transformations written in Java SE14 distributed over the different aspects of the transformation process compared to ATL?*

To answer these research questions, we devised a schema to translate 10 existing ATL transformations taken from the ATL Zoo[2] into Java. For comparing transformations specified in Java SE5 and SE 14, we use a combination of two metrics for measuring size and complexity, namely lines of code (LOC) and McCabe's cyclomatic complexity. Our comparison of complexity distributions is based on these metrics, and we incorporate the findings of Götz et al. on how complexity is distributed within ATL transformations (Götz and Tichy, 2020). For better understanding the differences revealed by our quantitative comparison, we conducted a qualitative analysis of corresponding code fragments written in the three different languages used in our study.

The remainder of this paper is structured as follows: First, Section 2 introduces the relevant aspects of ATL as well as the relevant differences between Java 5 and Java 14. In Section 3, we present our methodology for translating, reviewing and analysing the translated transformations. Afterwards, in Section 4, we give an overview of how ATL transformations were translated into Java. The results of our analysis and extensive comparison between the different transformation approaches are then presented in Section 5. Section 6 discusses potential threats to the validity of our work, while related work is considered in Section 7. Lastly, Section 8 concludes the paper and presents potential avenues for future research.

2 BACKGROUND

In this section, we briefly introduce the relevant background knowledge required for this paper. First, since model transformations can only be specified precisely based on some concrete model representation, we introduce the structural representation of models in MDE which is typically assumed by all mainstream model transformation languages, including ATL. Afterwards, since our work builds on ATL as well as the technological advancement of Java, it is necessary to introduce the relevant background knowledge on ATL and to present the important differences between Java SE5 and Java SE14, respectively.

2.1 Models in MDE

In MDE, the conceptual model elements of a modelling language are typically defined by a meta-model. The Eclipse Modeling Framework (EMF) (Steinberg et al., 2008), a Java-based reference implementation of OMG's Essential Meta Object Facility (EMOF) (OMG, 2016), has evolved into a de-facto standard technology to define meta-models that prescribe the valid structures that instance models of the defined modeling language may exhibit. It follows an object-oriented approach in which model elements and their structural relationships are represented by objects (EObjects) and references whose types are defined by classes (EClasses) and associations (EReferences), respectively. Local properties of model elements are represented and defined by object attributes (EAttributes). A specific kind of references are containments. In a valid EMF model, each ob-

[1]Interestingly, there was no significant evolution of the ATL language since its initial introduction in 2006 (Burgueno et al., 2019).

[2]https://www.eclipse.org/atl/atlTransformations

```
1 module NAME
2 create OUT1:MetaModelB, ...
3 [from|refining] IN1:MetaModelA, ...
4
5 [uses LIBRARY]*
6 [RULEDEF|HELPERDEF]*
```

Listing 1: Structure of an ATL module.

ject must not have more than one container and cycles of containments must not occur. Typically, an EMF model has a dedicated root object that contains all other objects of the model directly or transitively.

2.2 ATL

ATL distinguishes among three kinds of so-called *Units*, being either a *module*, a *library* or a *query*. Depending on the type of unit, they consist of *rules*, *helpers* and *attributes*. For data types and expressions, ATL uses the Object Constraint Language (OCL) (OMG, 2014).

2.2.1 Units

As illustrated in Listing 1, transformations are defined in *Modules*, taking a set of input models (line 3) which are transformed to a set of output models (line 2) by *rule* and *helper* definitions which make up the transformation (line 6).

Libraries: do not define transformations but only consist of a set of helper definitions. Libraries can be imported into modules to enhance their functionality (line 5).

Queries: are special types of libraries, that are used to define transformations from model elements to simple OCL types. They are comprised of a *query* element and a set of *helper* definitions.

2.2.2 Helpers and Attributes

Helpers allow outsourcing of expressions that can be called from within rules, similar to simple functions in general purpose languages. Helper definitions can specify a so-called *context* which defines the data type for which the helper is defined as well as parameters passed to the helper. ATL also allows the definition of *attribute* helpers. Attribute helpers differ from helpers in that they do not accept any parameter and always require a context data type. They serve as constants for the specified context. Listing 2 shows the syntax to define helpers and attribute helpers.

```
helper [context MODELTYPE]? def :
    NAME[(PARAMETERS)]? :TYPE  =
    EXPR;
```

Listing 2: Syntax to define Helpers.

2.2.3 Rules

In ATL, transformations of input models into output models are defined using *rules*. There are two main types of rules: *matched rules* and *called rules*.

Matched Rules: The declarative part of an ATL transformation is comprised by matched rules which are automatically executed on all matching input model elements, thus allowing to define type-specific transformations into output model elements. For this, the ATL engine traverses the input model in an optimized order. Furthermore, matched rules generate *traceability* links (trace links for short) between the source and target elements. These links can be navigated throughout the transformation specification to access references to elements created from a source element. Matched rules are comprised of four sections (see Listing 3):

- The *In-Pattern* (lines 2 to 3) defines the type of source model elements that are to be matched and transformed. An optional filter expression allows the definition of a condition that must be met for the rule to be applied.
- An optional *Using-Block* (lines 4 to 6) allows to define local variables based on the input element.
- The *Out-Pattern* (lines 7 to 10) then defines a number of output model elements that are to be created from the input element when the rule is applied. Each output model element is defined using a set of so-called *bindings* for assigning values to attributes of the output model element.
- Lastly, an optional *Action-Block* (lines 11 to 13) can be defined which allows the specification of imperative code that is executed once the target elements have been created.

Matched rules can also be defined as *lazy* rules by adding the keyword *lazy* to the rule definition (line 1). In contrast to regular matched rules, lazy rules are only executed when explicitly called for a specific model element that matches both the rule's type and its filter expression. They can be called multiple times on the same model element to produce multiple distinct output elements. To change the behaviour of lazy rules to always produce one and the same output element for the same source model element, lazy rules can be declared as *unique* (line 1).

Called Rules: As opposed to matched rules, called rules enable an explicit generation of target

```
[lazy| unique lazy]? rule NAME {
  from
  INVAR : MODELATYPE [(CONDITION)]*
  [using {
    [VAR : VARTYPE = EXPR;]+
  }]?
  to
  [OUTVAR : MODELBTYPE {
    [ATR <- EXPR,]+
  },]+
  [do {
    [STATEMENT;]*
  }]?
}
```

Listing 3: Syntax to define matched rules.

model elements in an imperative way. Called rules can be called from within the imperative code defined in the *Action-Block* of rules. They are defined similarly to matched rules. The main difference is that they do not contain an *In-Pattern* but instead allow the definition of required parameters. These parameters can then be used in the *Out-Pattern* and *Action-Block* to produce output model elements.

2.2.4 Refining Mode

The refining mode is a special execution mode for ATL modules which aims at supporting an easy definition of in-place transformations (Czarnecki and Helsen, 2006; Strüber et al., 2017). Normally, the ATL engine only creates new output model elements from input model elements matched by the rules defined in a module. However, in the refining mode, the ATL engine instead executes all rules on matching input elements and produces a copy of all unmatched input elements automatically. This aims to allow developers to focus solely on local modifications such as model refactorings rather than also having to manually produce copies of all other model elements.

2.3 Technological Advancements in Java14 Compared to Java5

Since the release of J2SE 5 in September of 2004, there have been a lot of improvements made to the Java language. In this section, however, we will only cover the ones relevant in the context of this paper. All the relevant features relate to a more functional programming style as they allow developers to express some key aspects of a transformation specification more concisely.

```
Function<Integer, Integer>
    doubleIt = (value) -> value *
    2;
```

Listing 5: Lambda expression definition based on Function.

2.3.1 Functional Interfaces

With the introduction of the *functional interfaces* in Java SE8, Java made an important step towards embracing the functional programming paradigm, paving the way to define lambda expressions in arbitrary Java code. Lambda expressions, also called anonymous functions, are functions that are defined without being bound to an identifier. This allows developers to pass them as arguments.

In essence, a *functional interface* is an interface containing only a single abstract method. This abstract method can then be implemented by means of a Java lambda expression (see Listing 5). One example of this is the interface called `Function<T,R>` (see Listing 4). It represents a function which takes a single parameter and returns a value.

```
public interface Function<T,R> {
  public R apply(T par);
}
```

Listing 4: Definition of the Function interface.

Lambdas defined with the interface `Function<T,R>` as their type are then nothing more than objects with their definition as the implementation of the apply method wrapped in a more functional syntax (see Listing 5).

Java provides a number of predefined functional interfaces, such as the aforementioned `Function<T,R>`, or `Consumer<T>` which takes one argument and has void as its return value.

2.3.2 Streams

Streams represent a sequence of elements and allow a number of different operations to be performed on the elements within the sequence. Stream operations can either be intermediate or terminal. This means that the operations can either produce another stream as their result or a non-stream result which therefore terminates the computation on the stream. This also means that intermediate operations work with all elements within the stream without the developer having to define a loop over it.

The example in Listing 6 shows how one can find and print all even numbers in a list using streams.

```
List<String> myList =
    Arrays.asList(1,2,3,4,5,6);
myList.stream().filter(i -> i % 2
    == 0).forEach(
    System.out::println);
```

Listing 6: Finding and printing all even numbers in a list.

Table 1: Meta-Data about the selected transformation modules.

Data	minimum	average	maximum	total
LOC	42	329	1125	13455
Rules	2	8	20	78
Helpers	0	9	74	82

3 METHODOLOGY

Our research methodology consists of three main subsequent steps. First, we selected transformation modules to translate and analyse. Afterwards, we developed a translation schema for translating ATL transformations into Java and translated the selected ones into both Java SE5 and Java SE14. Lastly, we selected and applied complexity measures on the Java and ATL code. The results were then analysed and compared in accordance with our research questions from Section 1.

3.1 Module Selection

The selection of ATL modules was done with several goals in mind. First we wanted to include transformations of different size and purpose. We also aimed to include both transformations using ATLs refining mode and normal transformations. Lastly, due to the fact that our translations would be done manually, we decided to limit the total number of transformations to 10 and the maximum size of a transformation to around 1000 LOC.

Since our work is, in part, based on the work presented in (Götz and Tichy, 2020) and their selection criteria align with ours, we opted to make the selection of modules from the set of transformations analysed by them.

This selection process resulted in a total of 10 ATL transformations [3], each of which stems from the well-known ATL Zoo. Basic meta-data about the transformations can be found in Table 1, while further details can be found in the ATL Zoo.

[3]A list with the names of the used transformations can be found under https://spgit.informatik.uni-ulm.de/stefan.goetz/javatransformationsv4/-/blob/master/resources/toCompare

3.2 Translation Schema Development and Application

To develop the translation schema, we followed the design science research methodology (Wieringa, 2014). We used the ATL solution found in the ATL Zoo for the families2persons case from the TTC'17 (Anjorin et al., 2017) as our initial test input for the translation scheme and focused on developing the schema for Java SE14.

The development process followed a simple, iterative pattern. A translation schema was developed and applied to the families2persons case. The resulting transformation was then reviewed, focusing on completeness and meaningfulness. Afterwards, the results of the review were used as input for reiterating the process.

In a final evolution step, the preliminary transformation schema was applied to all 10 selected ATL transformations. Afterwards, two researchers reviewed the resulting transformations separately based on a predefined code review protocol. In a joint meeting, the results of the reviews were discussed and final adjustments to the transformation schema were decided. These were then used to create a final translation of all 10 ATL transformations.

Lastly we ported the developed transformations to Java SE5 by forking the project, reducing the compiler compliance level and re-implementing the parts that were not compatible with older compiler versions.

To validate the correctness of the translated transformations, we used the input and output models that were provided within the ATL transformation projects. The input models were used as input for the Java transformations and the output models were then compared with the output of the transformations. If no output models were provided in the projects, we applied the ATL transformations to the input models and compared the Java results with the results generated by ATL. If no input model and output model were provided, we relied on the results of our code reviews. This validation approach is similar to how (Sanchez Cuadrado et al., 2020) validate their generated code.

3.3 Measure Selection and Analysis

Our analysis of the transformation specifications is guided by the research questions introduced in Section 1.

3.3.1 RQ1: How Much Less Complex Can Transformations be Written in Java SE14 Compared to Java SE5?

To compare the transformations written in Java SE14 and Java SE5, we decided to use code metrics focused on code complexity and size. For this reason, we chose McCabe's cyclomatic complexity as well as LOC which are shown to correlate with the complexity and size of software (Jabangwe et al., 2015). To keep the LOC count as fair as possible, we used the same standard code formatter for all Java code.

We applied the Java code metrics calculator (CK) (Aniche, 2015) on all 20 transformations to calculate both metrics. CK calculates metrics on the level of *classes*, *methods*, *fields* and *variables*. We opted to use the values calculated on the level of *methods*, i.e., the LOC and McCabe complexity of the method bodies because neither the *fields* level nor the *variables* level contained values for them and the *class* level would be too coarse-grained for our analysis. The metric values calculated by CK were then analysed and compared based on maximum, minimum, mean and average values.

3.3.2 RQ2,3: How Is the Complexity of Transformations Written in Java SE5/14 Distributed over the Different Aspects of the Transformation Process Compared to ATL?

The approach for these research questions is twofold and follows a top down methodology. First we compare the distribution of complexity within the Java code with regards to the different steps within the transformation process. In particular, we want to see how much effort needs to be put into writing those aspects that ATL can abstract away from. Afterwards, we focus on the actual code. Here we compare how code written in ATL compares to the Java code that represents the same aspect within a transformation.

To be able to analyse the complexity distribution in Java transformations, it is necessary to differentiate the different steps within the Java code. We therefore labelled each method in all transformations based on which aspects within the transformation process they represent. The relevant transformation aspects that were explicitly present in our Java implementations were:

- **Model Traversal:**, i.e., code that is associated with traversing the input model and selecting which 'rules' to apply to which element.
- **Element Creation and Tracing:**, i.e., code involved in creating output elements and trace links between source and target model elements.
- **Transformation:**, i.e., code that uses given data to populate the attributes of the output model elements.
- **Helper:**, i.e., code representing outsourced helper facilities.
- **Setup:**, i.e., code required to read/write models and to execute transformations on them.

```
rule SimpleBinding {
  from s : Member
  to t : Female (
    name <- s.firstName
  )
}
```

Listing 7: A rule with a simple binding.

```
rule Trace {
  from s : Member
  to t : Male (
    father <- s.familyFather
  )
}
```

Listing 8: A rule with a binding using traces.

We then used the metrics that were calculated for **RQ1** to create plots of the complexity distribution. The resulting plots were then compared with the results presented in (Götz and Tichy, 2020).

As for the code level comparison, we take a qualitative approach. We use a selection of three ATL fragments representing code which is often written in ATL transformations. The first fragment (see Listing 7) represents code that copies the value of an input attribute to an attribute of the resulting output model element, an action which constitutes 56% of all bindings in the set analysed by (Götz and Tichy, 2020). The second fragment (see Listing 8) represents code that requires ATL to use traceability links, which (Götz and Tichy, 2020) found to constitute 15% of all bindings. Because the attribute `s.familyFather` does not contain a primitive data type but a reference to another element within the source model, the contained value can not simply be copied to the output element. Instead ATL needs to follow the traceability link created for the referenced input element to find its corresponding output element which can then be referenced in the model element created from `s`. The last code fragment (see Listing 9) is a helper definition of typical size and complexity.

We use those code fragments and compare them with the Java code that they are translated to in order to highlight differences between the languages.

```
helper context Class def:
    associations:
    Sequence(Association) =
    Association.allInstances() ->
    select(asso | asso.value = 1);
```

Listing 9: A typical helper in ATL.

4 TRANSLATION SCHEMA

Our translation schema allows us to translate any ATL module into corresponding Java code. The only assumption we make is that all the meta-models of input and output models are explicitly available. The reason for this is that we work with EMF models in so-called static mode, which means that all model element types defined by a meta-model are translated into corresponding Java classes using the EMF built-in code generator.

As for the actual transformation, an ATL module is represented by a Java class which contains a single point of entry method that takes the root element of the input model as its input and returns an object of the root element of the output model. Additionally, some setup code is needed for extracting a model and its root element from a given source file, calling the entry point of the actual transformation class, and serializing the resulting output model.

Because traceability links need to be created before they can be used, we split the transformation process into two separate runs. The first run creates all target elements as well as all traceability links between them and their source elements, while the second run can safely traverse over model references and populate the created elements by utilizing the traceability links when needed. Consequently, the corresponding Java transformation class comprises two separate methods, dedicated to each run and being called by the entry point method. Listings 10 and 11 show an example of this translation.

For both model traversal as well as trace generation and resolving, we were able to develop generic libraries which can be reused across all transformation classes. The remainder of this section will first describe these two libraries in more detail, before we explain how the different types of rules and helpers of an ATL module are translated into the corresponding Java transformation class.

4.1 Traversal Library

The traversal library allows us to outsource the traversal of the source model and thus reduce the amount of boilerplate code written for each translated transformation. The traversal library builds upon a `HashMap` that maps a `Class<T>` to a `Consumer<EObject>`. The `Consumer<EObject>` interface represents a function that takes an input of type `EObject` and has a return type of `void`. During traversal, which is encapsulated within the library, each EObject can be matched with the key classes in the `HashMap` and the corresponding `Consumer` function can be executed.

```
module Example
create OUT: MetaModelB from IN :
    MetaModelA
...
```

Listing 10: Example ATL transformation.

```
public class MMA2MMB {
  public B transform(A root) {
    preTransform(root);
    return actualTransform(root);
  }
  private void preTransform(A
    root) {...}
  private void actualTransform(A
    root) {...}
}
```

Listing 11: Example translated Class.

This way, we only have to write code adding the required key-value-pairs to the traverser, while the code for traversing the input model as well as resolving the correct method which is to be called can be completely outsourced. Note that this is only necessary for *matched rules* since *lazy* and *called rules* are called within the transformation code and not automatically executed based on element type matching.

Since this setup relies on functional interfaces to work, it is only applicable in Java SE8 and greater. Therefore, in our transformations written in Java SE5, this whole process has to be reimplemented manually for each transformation module.

4.2 Trace Library

The trace library emulates the management of traceability links. Similar to the traversal library, the trace library is built based on a `HashMap`. In this case, however, the `HashMap` maps source `EObjects` to target `EObjects` and thus can be used both in Java SE5 and Java SE14.

In essence, the trace library exposes two methods. One for adding a trace, thus requiring the source and target objects to be passed as parameters. And one for resolving a trace based on a source object.

```
rule A2B {
  from
    a : A
  to
    b : B (bindings...)
}
```

Listing 12: Example ATL matched rule.

```
  public class Example {
    //...
    private void preTransform(A root) {
      //...
      preTraverser.add(A.class, a -> A2BPre(a));
    }
    private void actualTransform(A root) {
      //...
      traverser.add(A.class, a -> A2B(a));
    }

    private void A2BPre(A a) {
      TRACER.addTrace(a, new B());
    }
    private void A2B(A a) {
      B b = TRACER.resolve(a);
      //...bindings code
    }
  }
```

Listing 13: Example translated Class.

4.3 Matched Rule Translation

Matched rules are translated into two methods within the transformation class. One method is responsible for creating a target object and its corresponding trace link, and one method is responsible for populating the created target object in accordance with the bindings in its corresponding ATL rule. The second method will also incorporate all code corresponding to the imperative code written in the *Action-Block* of the translated rule. As already indicated in the beginning of this section when introducing our two-step transformation process, the main idea behind this separation is that all traces and referenced objects can be safely resolved within the second method (called during the second traversal) because they are created within the first method (called during the first traversal). That is, calls for the object and trace creation are then put into the body of the `preTransform` method, while calls for the second method are put into the body of the `actualTransform` method. This is illustrated by Listings 12 and 13. In Listing 13 the rule `A2B` from Listing 12 is translated into the methods `A2BPre` and `A2B`. `A2BPre` creates an empty `B` object as well as a trace between the input `A` and method `A2B` fills the corresponding `B` object with data as defined through the bindings from the `A2B` rule. To actually perform the transformation on all `A` objects the methods `preTransform` and `actualTransform` define for which type of object which method should be executed.

Lazy rules and unique lazy rules do not require this much overhead since they are called directly from within other rules/methods and thus do not need to be integrated into the traversal order. However, they do require traces to be crated and added to the global tracer. Additionally, methods translated from these types of rules have the target object as their return value rather than the return type being `void`.

4.4 Called Rule Translation

Called rules, much like lazy rules, can be translated into a single method that creates the output object and populates it in accordance with the bindings of the rule. Other than the methods created for matched rules, the methods for called rules can take more than one parameter as input since called rules in ATL can define an arbitrary amount of parameters of varying types.

4.5 Helper and OCL Expression Translation

Helpers can be translated into methods much like called rules. The contained OCL expressions can easily be translated into semantically equivalent Java code. One distinction that can be made here is again between the different Java versions used in terms of our study. Streams can be used to simulate the syntax of OCL, in particular the arrow symbol for implicitly navigating over collections), while older Java versions need to use loops instead. Table 2 shows a number of typical OCL expressions and their Java counterpart using streams.

Table 2: OCL expressions translated to Java streams.

OCL	Java
collection→select(e)	collection.stream().filter(e)
collection→collect(e)	collection.stream().map(e)
collection→includes(x)	collection.stream().anyMatch(a → x == a)
element.attribute	element.getAttribute()
i | i > 5	i → i > 5

5 RESULTS SUMMARY AND ANALYSIS

In this section, we present the results of our analysis in accordance with the research questions from Section 1.

5.1 RQ1: How Much Less Complex Can Transformations be Written in Java SE14 Compared to Java SE5?

Tables 3 and 4 summarise the calculated size (LOC) and complexity (McCabe) measurements on the method level for both the Java SE5 and Java SE14 transformation code.

Table 3: Size of the Java transformations in LOC.

Java	minimum	average	median	maximum
SE 5	3	10.4	7	135
SE 14	3	7.5	5	105

Table 4: McCabe's cyclomatic complexity of the Java transformations.

Java	minimum	average	median	maximum
SE 5	1	2.7	2	44
SE 14	1	1.5	1	11

As expected, the LOC required to write transformations in a newer Java version is less than in an older version. However, a reduction of about 30% on average is more than expected. This is confirmed by the median LOC which also shows about 30% reduction of Java SE14 compared to Java SE5.

The reduction in complexity is even more apparent for the cyclomatic complexity. On average, transformations written in Java SE14 are 45% less complex. The median again reflects this result. Furthermore, the maximum McCabe complexity is reduced from 44 to 11, which is a significant decrease.

Overall, the results reflect what was expected. Compared to Java SE5, new language features in Java SE14 help to reduce both the size and complexity of transformation specifications written in Java. However, the fact that the maximum cyclomatic complexity could be reduced by 75% is a surprising result.

5.2 RQ2: How Is the Complexity of Transformations Written in Java SE5 Distributed over the Different Aspects of the Transformation Process Compared to ATL?

Figure 1 shows a plot over the distribution of McCabe complexity split up into the different transformation aspects involved in a transformation written in Java SE5. It shows that about 73% of the complexity involved in writing a transformation in Java stems from the actual code representing the transformations and helpers. The other 27% are distributed among the model traversal, tracing and setup code. In ATL, it is these three aspects which do not have to be written explicitly but which are hidden behind ATL's syntax. In other words, this means that 27% of what is written in the Java transformation can be considered as overhead.

Moreover, a significant portion of the complexity of the transformation-related code stems from the implementation of helpers. About 44% of the transformation and helper complexity comes from the helper implementations. While this does not pose a problem directly, when comparing this with the results of (Götz and Tichy, 2020) where helpers only constitute about 16% of the complexity of an ATL module, it does raise some concern. The focus shifts away from the actual transformation code towards the outsourced helpers.

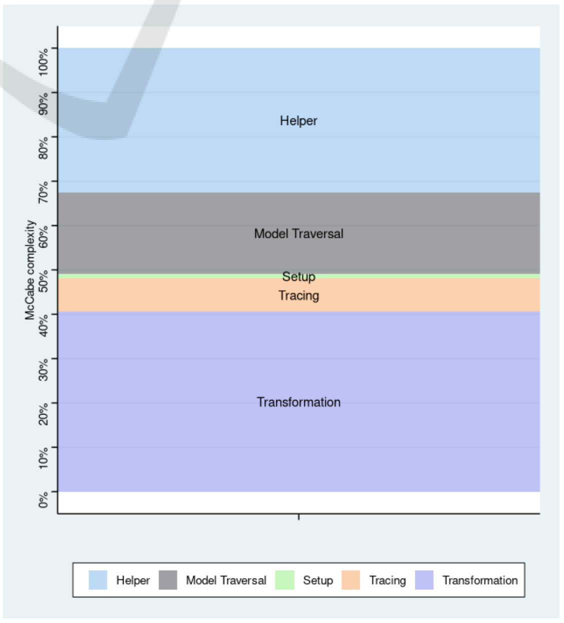

Figure 1: Distribution of McCabe complexity over transformation aspects in Java SE5.

Overall, the results support the consensus from back when ATL was introduced that a significant portion of complexity can be avoided when using a dedicated MTL for writing model transformations.

When comparing a simple binding (see Listing 7) written in ATL with its translation in Java SE5 (see Listing 14), there is not much difference. Both require nothing more than their language constructs for accessing attribute values and assigning them to a different attribute.

```
private void simpleBinding(Member
    s) {
  ...
  t.setName(s.getFirstName());
}
```

Listing 14: A rule with a simple binding in Java SE5.

This is not the case when traces are involved. While ATL allows developers to treat source elements as if they were their translated target element (see Listing 8), some explicit code needs to be written in Java (see Listing 15). As a result, the transformation specification gets more complex since it is not only required to call the trace resolution functionality, but it is also necessary to put some additional type information in so the Java compiler can handle the resulting object correctly. The type information is necessary since, as described in Section 4.2, the trace library holds `EObjects` which have to be converted to the correct type after they have been retrieved based on the source object.

```
private void simpleBinding(Member
    s) {
  ...
  t.setName(TRACER.resolve(
    s.familyFather, Male.class));
}
```

Listing 15: A rule with a binding using traces in Java SE5.

The increase in complexity is even more prevalent when looking at the translation of a typical helper. The helper in Listing 9 requires OCL code that works with collections which, thanks to OCL's "→ syntax", can be expressed in a concise manner. In Java SE5, however, as seen in Listing 16, the code gets a lot more complex and bloated. This is due to the fact that the only way to implement the selection is to iterate over the collection through an explicit loop (lines 3 to 9) and to use an if-condition within the loop (lines 4 to 6).

```
1 private List<Association>
      associations(Class self) {
2   List<Association> list = new
      LinkedList<Association>();
3   for (Association asso :
      ALLASSOCIATIONS) {
4     if (asso.getValue() == 1) {
5       list.add(asso);
6     }
7   }
8   return list;
9 }
```

Listing 16: A typical helper in Java SE5.

Overall, the examples show that simple bindings can be expressed easily in both ATL and Java SE5. Bindings involving trace resolution require some additional effort in Java SE5 while ATL can handle those like any other binding. The most significant difference, however, comes from expressions involving collections. Due to the required usage of explicit loops, the Java SE5 code blows up in size and complexity compared to the more compact ATL notation.

5.3 RQ3: How Is the Complexity of Transformations Written in Java SE14 Distributed over the Different Aspects of the Transformation Process Compared to ATL?

Given the observations from **RQ1** combined with the the general improvements that Java SE14 brings to the translation scheme, one would expect better results for the complexity distribution of transformations written in that Java version. However, when looking at Figure 2 which again shows a plot over the distribution of McCabe complexity split up into the different transformation aspects involved in a transformation written in Java, there is still a significant portion of complexity associated with the model traversal, tracing and setup code.

While the complexity associated with model traversal is greatly reduced by the use of the traversal library, the overall distribution between the actual code representing the transformations and helpers and the model traversal, tracing and setup code does not change much. About 24% of the overall transformation specification can still be considered as overhead code. Moreover, not only did this ratio stay similar compared to Java SE5, also the ratio between helper code complexity and transformation code complexity stayed about the same. Helper code makes up about 45% of the transformation and helper complex-

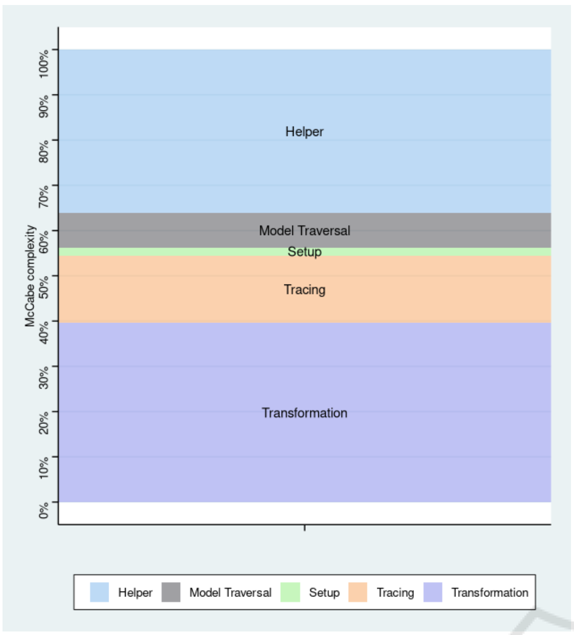

Figure 2: Distribution of McCabe complexity over transformation aspects in Java SE14.

ity. One potential reason for this is that while newer Java features help to reduce complexity, they do so for all aspects of the transformation, thus the distribution stays about the same.

The reason that the code related to trace management experiences an increase in its complexity ratio compared to other parts of the transformation can be explained by the fact that this code stayed the same between the different Java versions. Thus, while the complexity of all other components shrank, the complexity of trace management stayed the same, leading to higher relative complexity.

Overall, the results point towards even newer versions of Java still having to deal with the complexity overhead that ATL is able to hide. Specifically, while the traversal complexity could be greatly reduced through the use of newer language features, handling traces still entails a large overhead.

When comparing the code snippets for writing simple bindings and bindings involving traces in Java SE14 with ATL, there is no difference to the findings from comparing Java SE5 to ATL. This is due to the fact that no Java features introduced since SE5 help in reducing the complexity of code that needs to be written here.

Comparing the helper code snippets, however, does show some improvements of Java SE14 over Java SE5. Because of the introduction of the streams

```
private List<Association>
    associations(Class self) {
  return ALLASSOCIATIONS.stream()
  .filter(asso ->
    asso.getValue()==1)
  .collect(Collectors.toList());
}
```

Listing 17: A typical helper in Java SE14.

API, Java SE14 (see Listing 17) can now handle expressions involving collections nearly as seamless as ATL (see Listing 9). Only one key difference remains. Since collections in Java have to first be converted into streams and later reverted back into the collection type, some overhead still exists. In our example, these are the calls to `stream()` and `.collect(Collectors.toList())`. In principle, however, this difference could be eliminated by using an alternative GPL. The Scala programming language, for example, does not require a conversion between streams and collections.

Overall, the examples show that code for both simple bindings and bindings involving traces in Java SE14 stays just as complex in comparison to ATL as in Java SE5. Code involving collections, however, can now be expressed nearly as seamless as in ATL due to the introduction of the streams API in Java which offers a notation that is close to OCL notation.

6 THREATS TO VALIDITY

This section addresses potential threats to the validity of the presented work.

The transformations chosen for evaluation in our work were subject to a number of constraints. While we aimed to select a variety of transformation modules w.r.t. scope and size, the limitation of LOC and the source from which we selected the transformations may introduce a threat to the external validity of our work. The range of the domains involved in the transformations is diverse and should thus not threaten the external validity.

The next threat concerns the appropriateness and correctness of our translation schema and the resulting transformations. We tried to mitigate this threat by following the design science research method and using two separate reviewers for the proposed transformation schema. We also tested the correctness of the resulting transformations to the extent that was possible based on available resources.

Another threat related to the translation schema is that more than one way of translating ATL constructs into Java constructs and thus multiple translation schemas are possible. This impacts the conclusion validity of our study because different design decisions for the translation schema may impact the reproducibility of our results.

Lastly, the used metrics for measuring complexity should also be discussed. To prevent a bias due to the usage of a wrong measure, we opted to using two metrics for measuring complexity. Their suitability is supported by numerous publications in the literature as shown in (Jabangwe et al., 2015).

7 RELATED WORK

To the best of our knowledge, there exists no research that relates the complexity of transformations written in a MTL with that of transformations written in a GPL. However, there do exist several publications that provide relevant context for our work.

Hebig et al. investigate the benefit of using specialized model transformation languages compared to general purpose languages by means of a controlled experiment where participants had to complete a comprehension task, a change task, and they had to write one transformation from scratch (Hebig et al., 2018). They compare ATL, QVT-O and the GPL Xtend, and they found no clear evidence for an advantage when using MTLs. In comparison to their setup, we focus on a larger number of transformations. Furthermore, examples shown in the publication also suggest that they did not consider ATLs refining mode for their refactoring task nor did their examples focus on advanced transformation aspects such as tracing.

The authors of (Sanchez Cuadrado et al., 2020) propose A2L, a compiler for parallel execution of ATL model transformations. A2L takes ATL transformations as input and generates Java code that can be run from within their self-developed engine. Their data-oriented ATL algorithm describes how ATL transformations are executed in their code and closely resembles the structure that our translation schema follows.

The Simple Transformation Library in Java (SiTra) introduced in (Akehurst et al., 2006) provides a simple set of interfaces for defining transformations in Java. Their interfaces abstract rules and traversal in which they follow an approach similar to ours. However, they do not provide ways for trace management.

As previously described, parts of our research build upon the work presented in (Götz and Tichy, 2020). Here, the authors use a complexity measure for ATL proposed in literature to investigate how the complexity of ATL transformations is distributed over different ATL constructs such as matched rules and helpers. Their results provide a relevant data set to compare our complexity distributions in Java transformations to.

The authors of (van Amstel and van den Brand, 2011) use McCabe complexity to measure the complexity of ATL helpers. Among others, this is also done in (Vignaga, 2009). Similar to this, we use McCabe complexity on transformations written in Java, which includes translated helpers, to measure the complexity of the code.

The Model Transformation Tool Contest (TTC)[4] aims to evaluate and compare various quality attributes of model transformation tools. While some of these quality attributes (e.g., readability of a transformation specification) are related to the MTL used by the tool, most of the attributes are related to tooling issues (such as usability or performance) which are out of the scope of our study. Moreover, the contest is about comparing different MTLs with each other rather than comparing them with a GPL. Nonetheless, some cases have been presented along with a reference implementation in Java (Getir et al., 2017; Beurer-Kellner et al., 2020), which could serve as another source for comparing MTLs and GPLs more widely, including tooling- and execution-related aspects.

8 CONCLUSION

In this work, we presented how we developed and applied a translation schema to translate ATL transformations into Java. We also described our results of analysing the complexity and complexity distribution of these transformations. For this purpose, we used McCabe complexity as well as LOC metrics to measure the complexity of 10 transformations translated into Java SE5 and Java SE14, respectively. We analysed how the complexity between transformations written in those two Java versions differs as well as how the distribution of complexity compares to that in transformations written in ATL.

We found that new features introduced into Java since 2006 help to significantly reduce the complexity of transformations written in Java. However, we also showed that while the overall complexity of transformations is reduced, the distribution of how much of that complexity stems from code that implements functionality that ATL and other model transformation languages can hide from the developer stays

[4]https://www.transformation-tool-contest.eu

about the same. Thus, while the overall complexity is reduced with newer Java versions, the overhead entailed by using a general purpose language for writing model transformations still seems to be present.

For future work, we propose to also look at the transformation development process as a whole, instead of only at the resulting transformations. In particular, we are interested in investigating how the maintenance effort differs between transformations written in a GPL and those written in a MTL. For this purpose, the presented artefacts can be reused. Simple modifications to the ATL transformations can be compared to what needs to be adjusted in the corresponding Java code. Furthermore, because developers are the first to be impacted by the languages, it is also important to include users into such studies. For this reason, we propose to focus on user-centric study setups to be able to better study the impact of the language choice on developers.

Another potential avenue to explore is the comparison with a general purpose language that has a more complete support for functional programming such as e.g. Scala. Additional features such as pattern matching and easier use of functional syntax for translating OCL expressions into could potentially help to further reduce the complexity of the resulting transformation code.

REFERENCES

Akehurst, D. H., Bordbar, B., Evans, M. J., Howells, W. G. J., and McDonald-Maier, K. D. (2006). SiTra: Simple Transformations in Java. In *Model Driven Engineering Languages and Systems*, pages 351–364. Springer.

Aniche, M. (2015). *Java code metrics calculator (CK)*. https://github.com/mauricioaniche/ck.

Anjorin, A., Buchmann, T., and Westfechtel, B. (2017). The Families to Persons Case.

Beurer-Kellner, L., von Pilgrim, J., and Kehrer, T. (2020). Round-trip migration of object-oriented data model instances. In *Transformation Tool Contest at the Conference on Software Technologies: Applications and Foundations (TTC@STAF)*.

Burgueno, L., Cabot, J., and Gerard, S. (2019). The Future of Model Transformation Languages: An Open Community Discussion. *Journal of Object Technology*, 18(3):7:1–11. The 12th International Conference on Model Transformations.

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM systems journal*, 45(3):621–645.

Getir, S., Vu, D. A., Peverali, F., Strüber, D., and Kehrer, T. (2017). State Elimination as Model Transformation Problem. In *Transformation Tool Contest at the Conference on Software Technologies: Applications and Foundations (TTC@STAF)*, pages 65–73.

Götz, S. and Tichy, M. (2020). Investigating the Origins of Complexity and Expressiveness in ATL Transformations. *Journal of Object Technology*, 19(2):12:1–21. The 16th European Conference on Modelling Foundations and Applications (ECMFA 2020).

Götz, S., Tichy, M., and Groner, R. (2020). Claimed advantages and disadvantages of (dedicated) model transformation languages: a systematic literature review. *Software and Systems Modeling*.

Gray, J. and Karsai, G. (2003). An examination of DSLs for concisely representing model traversals and transformations. In *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*.

Hebig, R., Seidl, C., Berger, T., Pedersen, J. K., and Wasowski, A. (2018). Model Transformation Languages Under a Magnifying Glass: A Controlled Experiment with Xtend, ATL, and QVT. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, New York, NY, USA.

Jabangwe, R., Börstler, J., Šmite, D., and Wohlin, C. (2015). Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review. *Empirical Software Engineering*, 20(3):640–693.

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., and Valduriez, P. (2006). ATL: A QVT-like Transformation Language. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*.

Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). ATL: A model transformation tool. *Science of Computer Programming*.

Kehrer, T., Kelter, U., Ohrndorf, M., and Sollbach, T. (2012). Understanding model evolution through semantically lifting model differences with silift. In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 638–641. IEEE.

Kehrer, T., Taentzer, G., Rindt, M., and Kelter, U. (2016). Automatically deriving the specification of model editing operations from meta-models. In *International Conference on Theory and Practice of Model Transformations*, pages 173–188. Springer.

Krikava, F., Collet, P., and France, R. (2014). Manipulating Models Using Internal Domain-Specific Languages. In *Symposium On Applied Computing*, Gyeongju, South Korea.

OMG (2014). Object Constraint Language (OCL). https://www.omg.org/spec/OCL/2.4/PDF.

OMG (2016). Meta Object Facility (MOF). https://www.omg.org/spec/MOF.

Rindt, M., Kehrer, T., and Kelter, U. (2014). Automatic Generation of Consistency-Preserving Edit Operations for MDE Tools. *Demos@ MoDELS*, 14.

Sanchez Cuadrado, J., Burgueno, L., Wimmer, M., and Vallecillo, A. (2020). Efficient execution of ATL model transformations using static analysis and parallelism. *IEEE Transactions on Software Engineering*.

Schultheiß, A., Bittner, P. M., Kehrer, T., and Thüm, T. (2020a). On the use of product-line variants as experimental subjects for clone-and-own research: a case study. In *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*, pages 27:1–27:6. ACM.

Schultheiß, A., Boll, A., and Kehrer, T. (2020b). Comparison of Graph-based Model Transformation Rules. *Journal of Object Technology*, 19(2):3:1–21.

Sendall, S. and Kozaczynski, W. (2003). Model transformation: the heart and soul of model-driven software development. *IEEE Software*.

Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: eclipse modeling framework*. Pearson Education.

Strüber, D., Born, K., Gill, K. D., Groner, R., Kehrer, T., Ohrndorf, M., and Tichy, M. (2017). Henshin: A usability-focused framework for emf model transformation development. In *International Conference on Graph Transformation*, pages 196–208. Springer.

van Amstel, M. F. and van den Brand, M. (2011). Using Metrics for Assessing the Quality of ATL Model Transformations. In *MtATL@ TOOLS*.

Vignaga, A. (2009). Metrics for measuring ATL model transformations. *MaTE, Department of Computer Science, Universidad de Chile, Tech. Rep.*

Wieringa, R. J. (2014). *Design science methodology for information systems and software engineering*. Springer. 10.1007/978-3-662-43839-8.