Software and Systems Modeling (2022) 21:805–837
https://doi.org/10.1007/s10270-021-00937-3

REGULAR PAPER

Contrasting dedicated model transformation languages versus general purpose languages: a historical perspective on ATL versus Java based on complexity and size

Stefan Höppner[1] · Timo Kehrer[2] · Matthias Tichy[1]

Received: 17 June 2021 / Revised: 19 August 2021 / Accepted: 15 September 2021 / Published online: 17 November 2021

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 pssaradigm 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 and intuitive syntax. Nonetheless, MTLs are rarely adopted in practice, there is still no empirical evidence for the claim of easier development, 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 and size of model transformations written in three different languages, namely (i) the Atlas Transformation Language (ATL), (ii) Java SE5 (2004–2009), and (iii) Java SE14 (2020); the Java transformations are derived from an ATL specification using a translation schema we developed for 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 by as much as 45%. At the same time, however, the relative amount of complexity that stems from aspects that ATL can hide from the developer, which is about 40% of the total complexity, stays about the same. Furthermore we discovered that while transformation code in Java SE14 requires up to 25% less lines of code, the number of words written in both versions stays about the same. And while the written number of words stays about the same their distribution throughout the code changes significantly. Based on these results, we discuss the concrete advancements in newer Java versions. We also discuss to which extent new language advancements justify writing transformations in a general purpose language rather than a dedicated transformation language. We further indicate potential avenues for future research on the comparison of MTLs and GPLs in a model transformation context.

Keywords ATL · Java · Model transformations · Model transformation language · General purpose language · Comparison · MTL versus GPL · Historical perspective · Complexity measure · Size measure

1 Introduction

Model transformations are among the key concepts of the model-driven engineering (MDE) paradigm [1]. 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.

Communicated by Esther Guerra.

✉ Stefan Höppner
stefan.hoeppner@uni-ulm.de

Timo Kehrer
timo.kehrer@informatik.hu-berlin.de

Matthias Tichy
matthias.tichy@uni-ulm.de

[1] Ulm University, 89081 Ulm, Germany

[2] Humboldt University Berlin, 10099 Berlin, Germany

1.1 Context and motivation

In the literature, many advantages are ascribed to model transformation languages, such as better analysability, comprehensibility or expressiveness [2]. Moreover, model transformation languages aim at abstracting from certain recurring aspects of a model transformation such as traversing the input model or creating and managing trace information, claiming to hide complex semantics behind a simple and intuitive syntax [1,3–5].

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 [2]. Second, we argue that most of these claims emerged together with the first MTLs around 15 years ago. The Atlas Transformation Language (ATL) [6], 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 [7].

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 [8] and SERGe [9,10], or the implementation of model refactorings and model mutations in experimental setups of more recent empirical evaluations [11,12]. 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 [7], and, at least partially, by an empirical study conducted by Hebig et al. [13]. 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. There are also other features that certain model transformation languages can provide such as graph pattern matching, incrementality, bidirectionality or advanced analysis, but for now our study focuses solely on the abstraction and ease of writing argument.

1.2 Research goals and questions

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 one of its recent iterations (Java SE14) as well as at the level of 2006 (Java SE5) when ATL was introduced.[1] The goal of this approach is twofold. First, we intend to investigate how transformation code written in Java SE14 can be improved compared to the Java code using the Java version SE5 that was timely when ATL was released. Second, we want to contextualize these improvements by relating them to transformation code written in ATL. We opted to use both size and complexity measures for this purpose because both can provide useful insights for this discussion.

In order to achieve these goals, we developed four research questions to guide our research efforts:

RQ1 How much can the complexity and size of transformations written in Java SE14 be improved compared to Java SE5?

RQ2 How is the complexity of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to each other and ATL?

RQ3 How is the size of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to each other and ATL?

RQ4 How does the size of query aspects of transformations written in Java SE5 & SE14 compare to each other and ATL?

RQ1 aims to provide a general overview of how both size and complexity of transformations in Java might be improved using language features provided in newer Java versions. For a more detailed discussion and comparison it is then necessary to inspect and compare how the transformation code based is associated to the different aspects of a transformation, e.g. *model traversal*, *tracing* or the actual *transformation* of elements. This is the goal of **RQ2** and **RQ3** for complexity and size, respectively. With these two research questions we aim to investigate for which aspects new language features of Java help to reduce size and complexity of the associated code segments and what this means compared to ATL. Lastly, it is often assumed that querying aided by language constructs in MTLs is one key factor for their suitability over GPLs [14]. With **RQ4** we aim to investigate this assumptions via an explicit comparison between queries written in Java and ATL.

1.3 Research methodology

The process to answer the discussed research questions was structured around four consecutive steps. First, we selected a total of 12 existing ATL transformations taken from the

[1] Interestingly, there was no significant evolution of the ATL language since its initial introduction in 2006 [7].

Table 1 Meta-data about the selected transformation modules

Transformation name	LOC	# rules	# helpers
ATL2BindingDebugger	41	2	0
ATL2Tracer	96	2	0
DDSM2TOSCA	582	19	2
ExtendedPN2ClassicalPN	86	7	0
Families2Persons	49	2	2
istar2archi	99	6	1
Modelodatos2FormHTML	127	9	3
Palladio2UML	189	19	0
R2ML2XML	1125	60	1
ResourcePN2ResourceM	44	3	1
SimpleClass2RDBMS	63	4	3
UML22Measure	371	27	11
Average	236.25	13.3	2

ATL Zoo[2] and several projects from GitHub[3] to the basis for our study. References to all included transformations can be found in our supplementary material in Höppner, Tichy, and Kehrer [15]. 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 12 and the maximum size of a single transformation to around 1000 LOC. Since our work is, in part, based on the work presented in Götz and Tichy [16] and their selection criteria align with ours, we opted to make the selection of modules from the set of transformations analysed by them. The module selection process resulted in a total of 12 ATL transformations, from a variety of sources including the ATL Zoo. Basic meta-data about the transformations can be found in Table 1, while further details can be found in the supplementary materials.

Next, we devised, and tested, a schema to translate the selected ATL transformations to Java. To develop the translation schema, we followed the design science research methodology [17] using an iterative pattern for designing and enhancing the schema until it fit our purpose. 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 ATL transformations.

Afterwards, we developed a classification schema to divide Java code into its components and relate each component to the different aspects of the transformation process, i.e. transforming, tracing, and traversing. All Java code was then labelled based on the classification schema. For ATL, a similar schema from Götz et al. [16] already exists which we adopted and applied to the selected ATL transformations.

Lastly, we decided on and applied several code measures to allow us to compare the transformations. For comparing transformations specified in Java SE5 and SE 14, we use a combination of four metrics for measuring size and complexity, namely lines of code (LOC), word count (# words) [18], McCabe's cyclomatic complexity [19], and weighted method count (WMC). We use WMC based on McCabe complexity, i.e. the sum of the McCabe complexities of all elements, as the complexity measure in cases where the complexities of several elements need to be grouped together. Word count is used to supplement the standard code size measure LOC as a measure that is less influenced by code style and independent from keyword and method name size [18]. Furthermore, word count allows a direct size comparison between ATL and Java, which is hardly possible with LOC due to the languages' significantly different structure.

Our comparison of complexity and size distributions is thus based on LOC, word count, McCabe's cyclomatic complexity, and WMC, and we incorporate the findings of Götz et al. on how code is distributed within ATL transformations [16].

1.4 Results

Our analysis for **RQ1** shows that newer Java versions allow for a significant reduction in code complexity and lines of code, while the number of required words stays about the same. We attribute this to a more information dense style of writing single statements in the more functional programming style enabled by Java SE8 (2014) and newer.

The results for **RQ2** reflect the reduction in complexity overhead mainly in the methods involving model traversal. We also conclude that in newer language versions the most prominent remaining complexity overhead stems from manual trace management in Java compared to ATL.

The more detailed investigation done for **RQ3** supports these observations. We show that tracing is not only a prominent part in the methods dedicated to trace management but also in the methods that are dedicated to actually transforming input into output elements.

Overall the results for **RQ2** and **RQ3** suggest that still, a lot of complexity and size overhead for traversal, tracing, and supplementary code is required in Java even though newer Java versions improve the overall process of writing transformations. Of these, tracing is the biggest obstacle for efficiently developing transformations in a general purpose language. The overhead associated with this transformation aspect is the most significant and, arguably, most error-prone

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

[3] https://www.github.com.

one. A large portion of the advancements of Java SE14 over Java SE5 stem from the inclusion of more recent language aspects such as streams and functional interfaces. This fact is highlighted in our results from **RQ4** where those two aspects are the main factors for improvements in the size of OCL expressions written in Java.

1.5 Contributions and paper structure

This paper extends prior work on comparing Java and ATL transformations [20]. The extension consists of (i) a more detailed description of the applied translation schema from ATL to Java, (ii) the inclusion of an additional measure, namely number of words, for comparison, and (iii) the consideration of a larger set of transformations. Furthermore, we (iv) greatly expanded our discussion of overhead introduced by using Java for transformations based on the results from the newly included measure. This includes a more detailed inspection of Java code as well as a direct comparison between Java and ATL. Additionally, based on all the results and our own experiences, we (v) are now able to discuss more explicitly what newer Java versions improve over older ones and where the language is still lacking compared to ATL. Finally, we (vi) present a description of scenarios where these advancements are enough to justify Java over ATL and (vii) consider other features of model transformation languages not present in ATL and their impact on the suitability of general purpose languages.

The remainder of this paper is structured as follows: First, Sect. 2 introduces the relevant aspects of ATL as well as the relevant differences between Java 5 and Java 14. Afterwards, in Sect. 3, we give an overview of how we translate ATL transformations to Java. Because the discussions for **RQ2&3** require a precise classification of how code segments in Java are associated to the different transformation aspects, we provide an explanation for this in Sect. 4. In Sect. 5, we present our detailed method for analysing the size and complexity of the translated transformations. The results of our analysis and extensive comparison between the different transformation approaches are then presented in Sect. 6. Based on these results, Sect. 7 discusses our take-aways for what newer Java versions improve over older ones, where the language did not advance, and when these advancements are enough to justify Java over ATL. Section 8 then discusses potential threats to the validity of our work, while related work is considered in Sect. 9. Lastly, Sect. 10 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) [21], a Java-based reference implementation of OMG's Essential Meta Object Facility (EMOF) [22], has evolved into a de-facto standard technology to define meta-models that prescribe the valid structures that instance models of the defined modelling 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 object 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) [23].

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.

```
module NAME
create OUT1:MetaModelB, ...
[from|refining] IN1:MetaModelA, ...

[uses LIBRARY]*
[RULEDEF|HELPERDEF]*
```

List. 1 Structure of an ATL module.

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

List. 2 Syntax to define Helpers.

```
1  [lazy| unique lazy]? rule NAME {
2      from
3      INVAR : MODELATYPE [(CONDITION)]*
4      [using {
5          [VAR : VARTYPE = EXPR;]+
6      }]?
7      to
8      [OUTVAR : MODELBTYPE {
9          [ATR <- EXPR,]+
10     },]+
11     [do {
12         [STATEMENT;]*
13     }]?
14 }
```

List. 3 Syntax to define matched rules.

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.

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 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 [24,25]. 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

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

List. 4 Definition of the Function interface.

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

List. 5 Lambda expression definition based on Function.

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

List. 6 Finding and printing all even numbers in a list.

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 Java SE14 compared to Java SE5

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.

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. 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. This abstract method can then be implemented by means of a Java lambda expression (see Listing 5).

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.

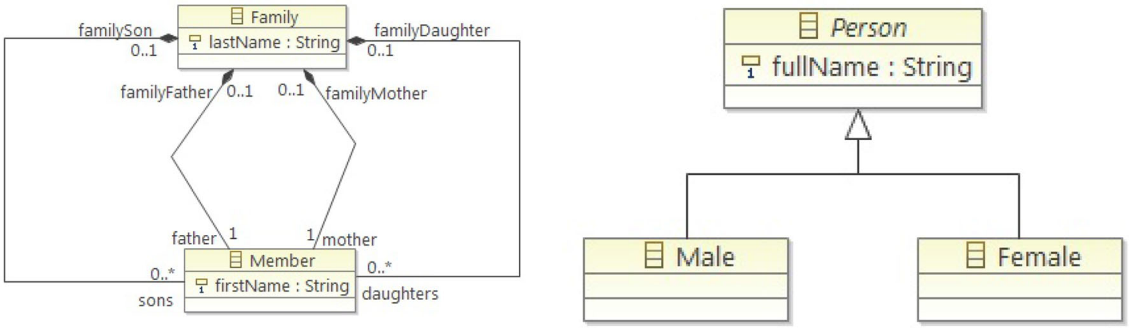
2.3.2 Streams

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.

3 Translation schema

In the following, we will present a detailed description of, first, how the translation schema was developed (see Sect. 3.1), before then describing the translation schema itself (Sects. 3.2 to 3.6).



(a) The Families meta-model.

(b) The Persons meta-model.

Fig. 1 The Families and Persons meta-models from the families2persons case taken from the ATL wiki [27]

The description of the translation schema is split into five parts. In Sect. 3.2, we describe the general setup used to emulate ATL semantics in Java and the basic structure that all translated modules follow. Then, in Sect. 3.3, we introduce and describe three libraries to reduce repetitive code between translated modules, one for trace handling, one for model traversal, and one for model loading and persisting. Sections 3.4 and 3.5 describe how the essential building blocks, namely matched rules and called rules, of ATL transformations are translated into Java. And lastly, in Sect. 3.6 we explain how helpers and general OCL expressions are translated.

All descriptions are illustrated by the use of a running example. For this, we use an ATL solution found in the ATL Zoo for the families2persons case from the TTC'17 [26] the code of which can be found in Listing 7, while its Java SE14 counterpart can be found in Listing 8. The meta-models for the transformation case are shown in Fig. 1. The example illustrates how different ATL elements are translated into their corresponding Java code based on the described schemata. Our descriptions will focus on the Java SE 14 translation schemata. Notable differences between the Java SE 14 and Java SE5 translation schemata are highlighted as such.

3.1 Schema development

To develop the translation schema, we followed the design science research methodology [17]. We used the ATL solution found in the ATL Zoo for the families2persons case from the TTC'17 [26] 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 by the main author and applied to the Families2Persons case. The resulting transformation was then reviewed by one co-author, 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 12 selected ATL transformations. Afterwards, both co-authors 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 12 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. Since neither an input nor an output model was available for the R2ML2XML transformation, we had to rely solely on the results of our code reviews for its validation. This validation approach is similar to how Sanchez Cuadrado et al. [28] validate their generated code.

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.

3.2 General setup and module translation

In our translation scheme, we generally assume that each model contains a single root element. This is standard for EMF but could be easily extended by using lists as input and output.

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 the root element of the output model. The `transform` method in line *13* of Listing 8 represents this entry point for the families2persons transformation. It takes the *root* model element of type `Family` from the input model and returns a List of type `Person` which serves as the root element for the `Persons` meta-model.[4]

```
module Families2Persons;
create OUT : Persons from IN : Families;

helper context Families!Member def: familyName :
     String =
    if not self.familyFather.oclIsUndefined() then
        self.familyFather.lastName
    else
        if not self.familyMother.oclIsUndefined()
     then
            self.familyMother.lastName
        else
            if not self.familySon.oclIsUndefined()
        then
                self.familySon.lastName
            else
                self.familyDaughter.lastName
            endif
        endif
    endif;

helper context Families!Member def: isFemale()
        : Boolean =
    if not self.familyMother.oclIsUndefined() then
        true
    else
        if not
     self.familyDaughter.oclIsUndefined()
        then
            true
        else
            false
        endif
    endif;

rule Member2Male {
    from
        s : Families!Member (not s.isFemale())
    to
        t : Persons!Male (
    fullName <- s.firstName + ' ' + s.familyName
    )
}

rule Member2Female {
    from
        s : Families!Member (s.isFemale())
    to
        t : Persons!Female (
    fullName <- s.firstName + ' ' + s.familyName
    )
}
```

List. 7 Families2Persons ATL solution.

[4] In reality the `Persons` meta-model does not have a root element and the list is used as a substitute for the transformation to conform with the translation schema as well as general EMF standards. To produce this list from the transformed elements the `Family2List` method in lines *36-42* is introduced which does not have a counterpart in ATL.

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. The code required for our running example is shown in Listing 9. We utilize one of our developed libraries, namely `IO`, for reading an xmi-file containing a `Families` model, extracting the root object of type `Family` and passing it to the `transform` method of the `Families2Persons` class to initiate the actual transformation. The resulting output of type `List<Person>` is then written to an xmi-file, again, utilizing our `IO` library.

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. In our example in Listing 8, the methods `preTransform(Family root)` and `actualTransform(Family root)` in lines *18* and *25* represent these two runs. Their implementation will be explained later throughout Sect. 3.4.

3.3 Libraries

For both model traversal as well as trace generation and resolving, we developed generic libraries which can be reused across all transformation classes. Additionally, we also required a library to outsource the reading and writing of models from and into files. The remainder of this section will describe these libraries in more detail.

3.3.1 IO library

The IO library contains methods used for reading and writing models from and to files. The library exposes two methods, namely `readModel(String uri)` and `persistModel(EObject root, String uri)` which both bundle together several EMF and file-IO methods to achieve the desired effects. To do so the library utilizes the `Resource`[5] type which represents a "persisted document" in EMF and allows to read and write `EObjects` from and to it. To be able to read and write different file types such as *xmi* or *ecore*, a corresponding `ResourceFactory` needs to be registered in the *ExtensionToFactoryMap* of the ResourceFacotry registry. For this reason, we opted to only support *xmi*, *ecore* and *uml* files since EMF provides default ResourceFactory implementations for all three.

The `persistModel` method takes a root element of a model as well as a desired output path, and creates a resource containing the root element (and all its children) which is then saved to the specified path. The `readModels` method reverses this approach by extracting the resource pointed to by the passed path and returning all contents of the referenced resource to the caller. Due to the makeup of EMF compliant files such as *xmi*, *ecore* or *uml* the first element within the contents will then always contain the root element of the model within the file which can then be used as seen in Listing 9.

3.3.2 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. It builds upon a `HashMap` that maps a `Class<?>` to a `Consumer<EObject>`. The `Consumer<EObject>` interface represents a function that takes an input object of type `EObject` and has a return type of `void`. During traversal, which is encapsulated within the library, the Consumer function that corresponds to an `EObject` can be retrieved from the `HashMap` by using the class of the EObject as key. To achieve this, the library exposes the methods `addFunction` and `traverseAndAccept`.

The `addFunction` method allows us to add a key-value-pair to the encapsulated hashmap. The `traverseAndAccept` method then takes an `Iterable` collection containing `EObjects`, iterates over all contained objects, fetches the function that corresponds to the concrete class of the `EObject`, and executes it. This way, we only have to write code that adds the required key-value-pairs to the traverser, while the code for traversing the input model as well as resolving the correct function which is to be called is completely outsourced. Note that adding such function calls 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. An example of how the traversal library is used can be found in lines *19-22* and *28-31* of Listing 8 and will be explained in more detail in Sect. 3.4.

For the Java SE5 solution we decided on an alternative solution using the conditional dispatcher pattern instead of outsourcing the traversal. The reason for this was a weighing of alternatives. Outsourcing the traversal in Java SE5 would require the utilisation of anonymous classes. This in turn would offer a similar workflow and an equal McCabe complexity for defining model traversal as with the functional interface solution in Java SE14. It would however significantly increase the required number of words and lines of code compared to the conditional dispatcher solution. Only

[5] https://download.eclipse.org/modeling/emf/emf/javadoc/2.4.3/org/eclipse/emf/ecore/resource/Resource.html.

```
public class Families2Persons {
   private static final PersonsFactory PERSONSFACTORY = PersonsFactory.eINSTANCE;
   private static final Tracer TRACER = new Tracer();

   private static boolean isFemale(Member member) {
     return member.getFamilyDaughter() != null || member.getFamilyMother() != null;
   }

   private static String familyName(Member member) {
      return ((Family) member.eContainer()).getLastName();
   }

   public static List<Person> transform(Family family) {
      preTransform(family);
      return actualTransform(family);
   }

   private static void preTransform(Family root) {
      var iterator = root.eAllContents();
      var traverser = new Traverser(TRACER);
      traverser.addFunction(Member.class, x -> {Member2MalePre((Member) x);Member2FemalePre((Member) x);});
      traverser.traverseAndAcceptPre(iterator);
   }

   private static List<Person> actualTransform(Family root) {
      var newRoot = Family2List(root);

      var iterator = root.eAllContents();
      var traverser = new Traverser(TRACER);
      traverser.addFunction(Member.class, x -> {Member2Male((Member) x);Member2Female((Member) x);});
      traverser.traverseAndAccept(iterator);

      return newRoot;
   }

   private static List<Person> Family2List(Family root) {
      var persons = new LinkedList<Person>();
      persons.add(TRACER.resolve(root.getFather(), Male.class));
      persons.add(TRACER.resolve(root.getMother(), Female.class));
      persons.addAll(root.getDaughters().stream().map($ -> TRACER.resolve($, Female.class)).collect(Collectors.toList()));
      persons.addAll(root.getSons().stream().map($ -> TRACER.resolve($, Male.class)).collect(Collectors.toList()));
      return persons;
   }

   private static void Member2MalePre(Member m) {
      if (!isFemale(m)) {
         TRACER.addTrace(m, PERSONSFACTORY.createMale());
      }
   }
   private static void Member2Male(Member m) {
      var t = TRACER.resolve(m, PERSONSFACTORY.createMale());
      t.setFullName(m.getFirstName() + " " + familyName(m));
   }

   private static void Member2FemalePre(Member m) {
      if (isFemale(m)) {
         TRACER.addTrace(m, PERSONSFACTORY.createFemale());
      }
   }
   private static void Member2Female(Member m) {
      var t = TRACER.resolve(m, PERSONSFACTORY.createFemale());
      t.setFullName(m.getFirstName() + " " + familyName(m));
   }
}
```

List. 8 The Families2Persons solution translated in Java SE14.

with the improved syntax provided through the functional interfaces in Java SE8 could a decrease of the McCabe complexity be accompanied with an uniform word count and lines of code. Overall, the decision leads to an increase in the McCabe complexity of the traversal code in Java SE5 but allows for word count and LOC to remain stagnant. We will come back and discuss the impact of this decision (in the relevant parts of our results discussionl in Sect. 7.1) later on.

This design decision affects the methods `preTransform` and `actualTransform`. Their implementation in Java SE5 is shown in Listing 10. Instead of populating the traverser objects we instead manually iterate over the whole model and decide which methods to call based on the type of the currently visited object.

```
List<EObject> ins =
     IO.readModel(''Family.xmi'');
Family family = (Family) ins.get(0);
List<Person> persons =
     Families2Persons.transform(family);
IO.persistModel(persons, ''persons.xmi'');
```

List. 9 Setup code for the Families2Persons transformation.

```
//...
private static void preTransform(Family root) {
    TreeIterator<EObject> iterator =
     root.eAllContents();
    while (iterator.hasNext()) {
        EObject next = iterator.next();
        if (next instanceof Member) {
            Member m = (Member) next;
            Member2MalePre(m);
            Member2FemalePre(m);
        }
    }
}

private static List<Person>
     actualTransform(Family root) {
    List<Person> newRoot = Family2List(root);

    TreeIterator<EObject> iterator =
     root.eAllContents();
    while (iterator.hasNext()) {
        EObject next = iterator.next();
        if (next instanceof Member) {
            Member m = (Member) next;
            Member2Male(m);
            Member2Female(m);
        }
    }
    return newRoot;
}
//...
```

List. 10 Translated model traversal in Java SE5.

3.3.3 Trace library

The trace library emulates the management of traceability links of ATL. 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. First, for adding a trace (`addTrace`), thus requiring the source and target objects to be passed as parameters. Second, for resolving a trace based on a source object named `resolve`. To achieve type consistency `resolve` also requires the class of the intended target object to be passed as parameter. An example of how the trace library is used can be found in line *51* of Listing 8 and will be explained in more detail in Sect. 3.4.

For more advanced trace management, additional methods exist that take an additional String parameter to be able to add and distinguish multiple target objects for a single source object. This functionality is sometimes required to access not the direct target object but another object that was created during the translation of a source object.

3.4 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 Sect. 3.2 when introducing our two-step transformation process, the main idea behind this separation is that all traces and referenced objects can be safely resolved by the second method (called during the second traversal) because they are created by the first method (called during the first traversal). That is, calls for the object and trace creation are put by the `preTransform` method, while calls for the second method are put into the body of the `actualTransform` method.

For the rules `Member2Male` and `Member2Female`, this is illustrated in lines *45* and *50* of Listing 8. The rule `Member2Male` from Listing 7 is translated into the methods `Member2MalePre` (in line *45* of Listing 8) and `Member2Male` (in line *50* of Listing 8). `Member2MalePre` creates an empty `Male` object as well as a trace from the input `Member`, and method `Member2Male` fills the corresponding `Male` object with data as defined through the bindings from the `ATL` rule. To actually perform the transformation on all `Member` objects, the methods `preTransform` and `actualTransform` define for which type of object which method should be executed. This is done using methods from

```
private static Female lazyMember2Female(Member
     m) {
    if (isFemale(m)) {
        Female t = TRACER.add(m,
    PERSONSFACTORY.createFemale());
        t.setFullName(m.getFirstName() + " " +
    familyName(m));
        return t;
    }
    return null;
}
```

List. 11 Example translated lazy rule.

```
private static B uniqueLazyMember2Female(A a) {
    Female t = TRACER.resolve(m,
     PERSONSFACTORY.createFemale());
    if (t == null) {
        if (isFemale(m)) {
            t.setFullName(m.getFirstName() + "
     " + familyName(m));
            return t;
        }
        return null;
    }
    return t;
}
```

List. 12 Example translated unique lazy rule.

the traversal library to add the corresponding function calls for the `Member` class as shown in lines *21* and *30* of Listing 8.

A special feature that comes from using our traversal library is that we only need to translate the condition whether a rule should be applied in the pre method that is translated from it. This is because the `traverseAndAccept` method only executes the corresponding function for an object after it verified that an associated target object can be found via a trace. If no target object can be found, the function is not executed. An example of this can be found in the translation of the `Member2Male` rule. Line *32* of Listing 7 states that `Member2Male` is only executed under the condition that `not s.isFemale()`. In the Java code in Listing 8, this is only translated into the `Member2MalePre` method in line *46*, whereas `Member2Male` in line *50* does not contain this condition.

Lazy rules and unique lazy rules do not require as much overhead as matched rules 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 created 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`. Suppose `Member2Female` was a lazy matched rule. In that case, instead of the code in lines *21*, *30*, and *59-63* for `Member2Female`, only the code shown in Listing 11 would be added to the `Families2Persons` class. The method `lazyMember2Female` returns an object of type `Female` while also creating a trace from the passed `Member` to the returned `Female`. In case `Member2Female` was a unique lazy matched rule, a precondition using trace links is added to the translated Java code that ensures that the method always returns the same object when called for the same input object. This is illustrated in Listing 12.

3.5 Called rule translation

Called rules, much like lazy rules, can be translated into a single method that creates the output object, populates it in accordance with the bindings of the ATL rule, and then returns it. 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. Moreover, called rules do not create or use trace links. A sample called rule translated into Java can be found in Listings 13 and 14.

```
rule calledMember2Female(Member m, String name)
     {
    to
        t : Female (
            fullName <- name
        )
}
```

List. 13 Example ATL called rule.

```
private static Female
     calledMember2Female(Member m, String name) {
    Female t = PERSONSFACTORY.createFemale();
    t.setFullName(name);
    return t;
}
```

List. 14 Example translated called rule.

3.6 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. Examples of such semantically equivalent translations can be found in lines *9-11* of Listing 8 which correspond to the OCL code in lines *4-17* of Listing 7. 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 OCL expressions and their Java SE14 counterpart using streams. Note that in contrast to OCL, Java requires all collections to be converted to streams and back to be able to manage them in a func-

Table 2 A selection of OCL expressions translated to Java SE14

OCL	Java SE14
collection->select(e)	collection.stream().filter(e).collect(Collectors.toCollection())
collection->collect(e)	collection.stream().map(x -> e.apply(x)).collect(Collectors.toCollection())
collection->includes(x)	collection.stream().anyMatch(a -> x == a).collect(Collectors.toCollection())
element.attribute	element.getAttribute()
collection.attribute	collection.stream().map(x -> x.getAttribute()).collect(Collectors.toCollection())
i | i > 5	i -> i > 5

tional programming style. The same expressions written in Java SE5 without streams can be found in Listings 24 to 29 in Appendix A.

4 Code classification schema

In this section we introduce the classifications of Java and ATL code used throughout **RQ2** and **RQ3**. The ATL classification described in Sect. 4.1 is taken from [16] and is based on the hierarchical structure of ATL. The classification of Java code described in Sect. 4.2 was developed specifically for the analysis of this research. It is based in the structure of Java code and its components as well as the relation thereof to general transformation aspects and ATL. We will again use the families2persons example to illustrate how the classification schemas are applied.

4.1 ATL

The hierarchy for the ATL classification was already established by Götz and Tichy [16] and consists of the following levels and their corresponding categories:

1. Module Level
2. Rule Type & Helper Level
3. Rule Blocks Level
4. Content Level
5. Binding Level

The aim of this classification system is to differentiate the different components and their contained subcomponents within an ATL module. As such, this classification represents a way to indicate how a syntax element is contained within the complete structure of the ATL code. This allows us to make precise observations on the structure of ATL modules based on their components and, for example, the distribution of number of words required to write each component. An overview of the classification hierarchy can be found in Fig. 2. And the complete labelling for the ATL solution of Families2Persons can be found in Fig. 3.

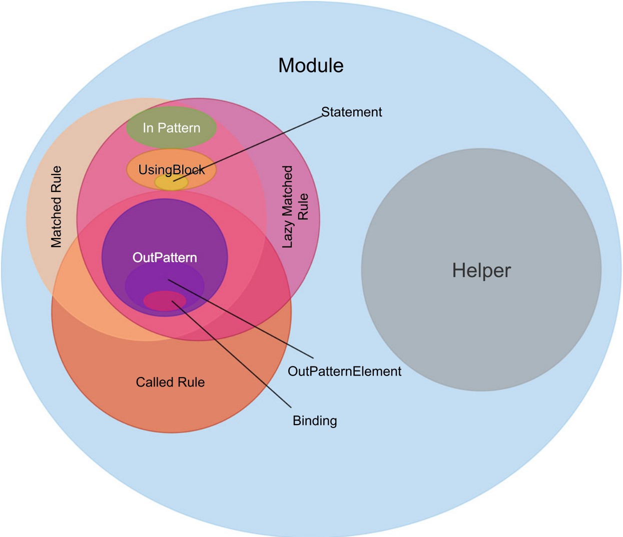


Fig. 2 Overview of the ATL classification from Götz and Tichy ([16]

The **Module Level** defines the belonging of all elements within a module to said module. Below it on the **Rule Type & Helper Level** a distinction between helpers and the different types of rules is made. In the Families2Persons example from Listing 7 and Fig. 3 the helpers in lines *4* & *19* are labelled as *Helper*, while both rules `Member2Male` and `Member2Female` in lines *30* & *39* are labelled as *Matched Rule*. All elements within the rules and helpers again inherit the respective classification for this level from their parent elements.

The **Rule Blocks Level** distinguishes between the different types of blocks that make up rules, i.e. *Using Block*, *OutPattern*, *InPattern*, and *Action Block*. A more specific distinction of helper contents is not done due to them only containing OCL expressions. The rules in the Families2Persons example only contain InPatterns (lines *31-32, 40-41*) and OutPatterns (lines *33-36, 42-45*).

Below the Rule Blocks Level the **Content Level** then allows a more precise description of the elements contained within the rule blocks. The potential classifications on this level are: *OutPatternElement*, *Statement*, and *Vari-*

able Declaration. Lines *43-45* for example are labelled as an *OutPatternElement*.

Lastly, the **Binding Level** again only contains one characteristic and allows to label bindings as exactly that. Lines *35* and *44* are bindings and thus labelled as such as seen in Fig. 3.

4.2 Java

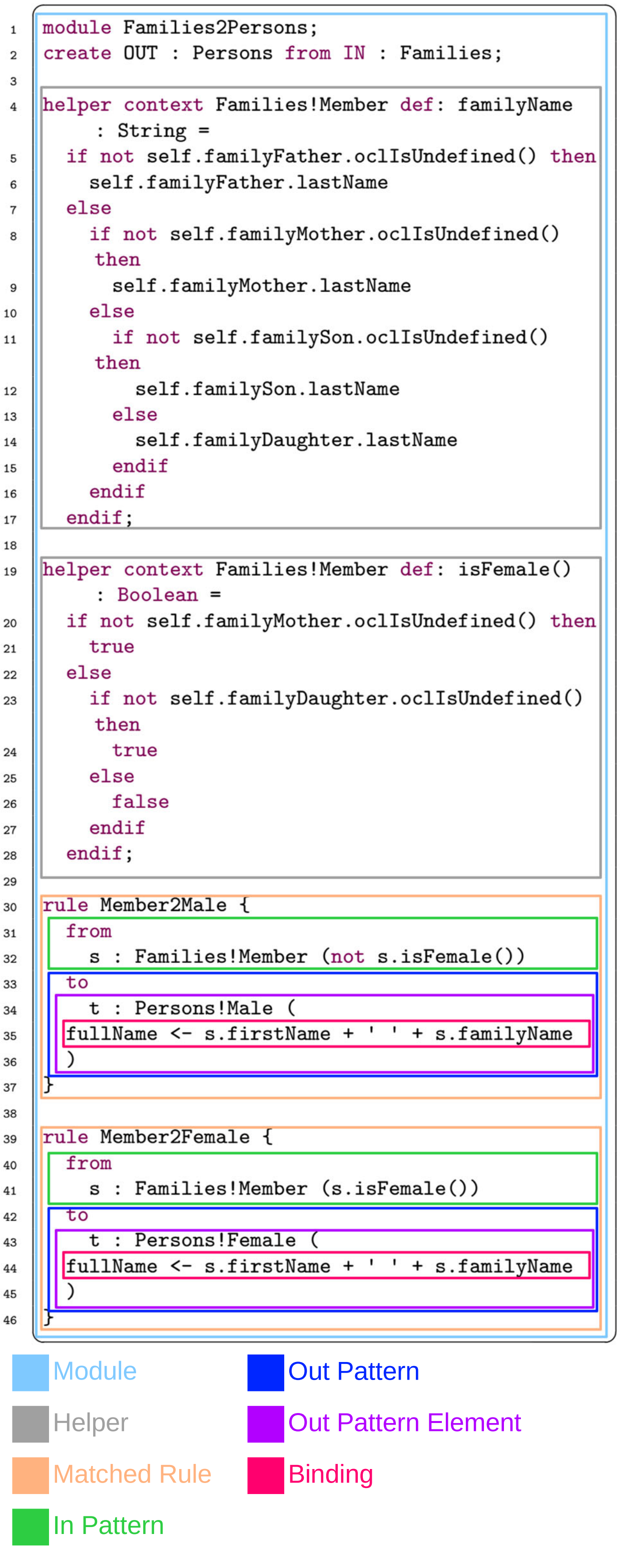
In order to draw parallels between transformation code written in Java and ATL, it is necessary to relate all code components in the Java code to the transformation aspects they implement. For this purpose, we developed a hierarchical classification for Java code. The hierarchy follows the natural structure of Java code much like the classification for ATL. However, contrary to ATL, the code structure of Java does not allow us to directly break it down into transformation-related components. This is due to the fact that Java is focused around object-oriented and imperative components rather than transformation-specific ones. As a result, the classification schema breaks Java code down into its OO and imperative components and then relates those components to transformation aspects. The hierarchy levels of the classification are as follows:

1. Class Level
2. Attribute & Method Level
3. Statement-Type Level
4. ATL Counterpart Level

An overview of the classification levels and the characteristics attributed to each level can be found in Fig. 4. A sample labelling for the Java solution of Families2Persons can be found in Fig. 5.

The **Class Level** stands on top of the hierarchy. The class level itself is made up of only one type of characteristic, the *Class* itself. In the Families2Persons example from Listing 8 the class definition and all elements contained within the class body is thus labelled as belonging to the class characteristic of the Class Level (as seen in Fig. 5). This also indirectly represents a relation between the class and the transformation module from which it was translated from, indicating that the class and all its components relate to the transformation module and its components. More specific relation between the contained components is then described through the lower levels within the classification system.

Below the Class Level lies the **Attribute & Method Level** in which we classify to which transformation aspect an attribute or method is related. The characteristics that can be attributed on this level are: *Traversal* when a method is used for the traversal of the input model. *Transformation* when a method contains code for the actual transformation of one model element to another. *Tracing* for all



```
1  module Families2Persons;
2  create OUT : Persons from IN : Families;
3
4  helper context Families!Member def: familyName
       : String =
5    if not self.familyFather.oclIsUndefined() then
6      self.familyFather.lastName
7    else
8      if not self.familyMother.oclIsUndefined()
        then
9        self.familyMother.lastName
10     else
11       if not self.familySon.oclIsUndefined()
        then
12         self.familySon.lastName
13       else
14         self.familyDaughter.lastName
15       endif
16     endif
17   endif;
18
19 helper context Families!Member def: isFemale()
       : Boolean =
20   if not self.familyMother.oclIsUndefined() then
21     true
22   else
23     if not self.familyDaughter.oclIsUndefined()
        then
24       true
25     else
26       false
27     endif
28   endif;
29
30 rule Member2Male {
31   from
32     s : Families!Member (not s.isFemale())
33   to
34     t : Persons!Male (
35   fullName <- s.firstName + ' ' + s.familyName
36   )
37 }
38
39 rule Member2Female {
40   from
41     s : Families!Member (s.isFemale())
42   to
43     t : Persons!Female (
44   fullName <- s.firstName + ' ' + s.familyName
45   )
46 }
```

Fig. 3 Labelled ATL solution for the Families2Persons case

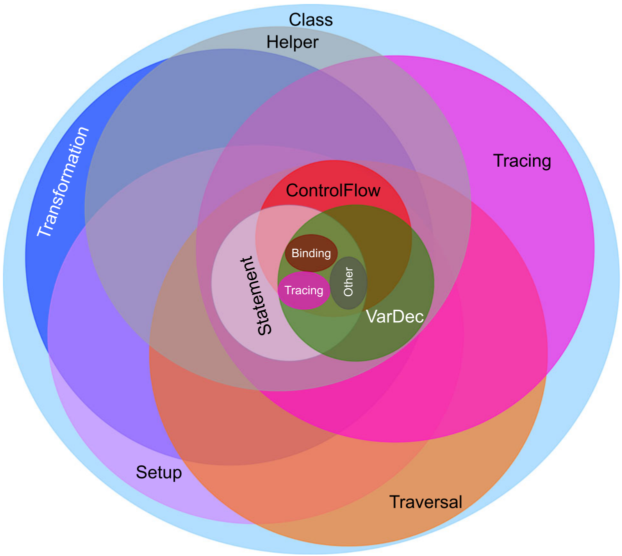


Fig. 4 Overview of the makeup of our Java classification

methods that are related to the creation or resolution of traces. *Helper* when a method corresponds to a helper and lastly *Setup* for all attributes that are required to exist for access throughout the transformation. The `isFemale` method in lines *5-7* from Fig. 5 is thus assigned the label *Helper* for the **Attribute & Method Level** in addition to its *Class* label on the **Class Level**. The `transform`, `preTransform` and `actualTransform` methods all get assigned the *Traversal* label, while `Family2List`, `Member2Male` and `Member2Female` are all labelled as *Transformation* related on the Attribute & Method Level. Lastly, `Member2MalePre` and `Member2FemalePre` both relate to *Tracing* and are thus characterized as such. All statements within the methods again inherit the classification of the Class Level and the Attribute & Method level from their respective parents in which they are contained in and get more specialized again through the lower levels within the system.

Below the Attribute & Method Level then lies the **Statement-Type Level** in which all statements within methods are characterized based on whether they are *Control Flow* statements (i.e. conditions or loops), *Variable Declaration*s or any other type of *Statement*. The categorization on this level does not directly relate to any transformation aspect but rather allows us to differentiate between different types of statements in Java that are relevant for highlighting differences between the structure of Java and ATL code. The condition defined in line *46* of Fig. 5 is labelled as belonging to *Control Flow* on this level while again inheriting its Class Level and Attribute & Method Level from its container Method `Member2MalePre`.

The next lower level is the **ATL counterpart Level**. On this level, we categorize whether a statement fulfils the role of a *Binding* in ATL or if it contains code to create or resolve *Traces* or if it is any *Other* type of Java code that does not directly relate to transformation aspects. At this level, one would expect that the categorization of statements is dependent on the categorization of the **Attribute & Method Level** of the methods they are contained in, i.e. a statement within a *Transformation* method should either be categorized as *Binding* or *Other*. However, in Java transformations these boundaries become somewhat blurred due to the fact that traces need to be explicitly resolved to access the corresponding output model elements when assigning them to output attributes. This can for example be seen for line *51* of Fig. 5. The classification comes from it being a variable declaration that assigns the result of a trace resolution call within a method that performs the transformation of a `Member` into a `Male`.

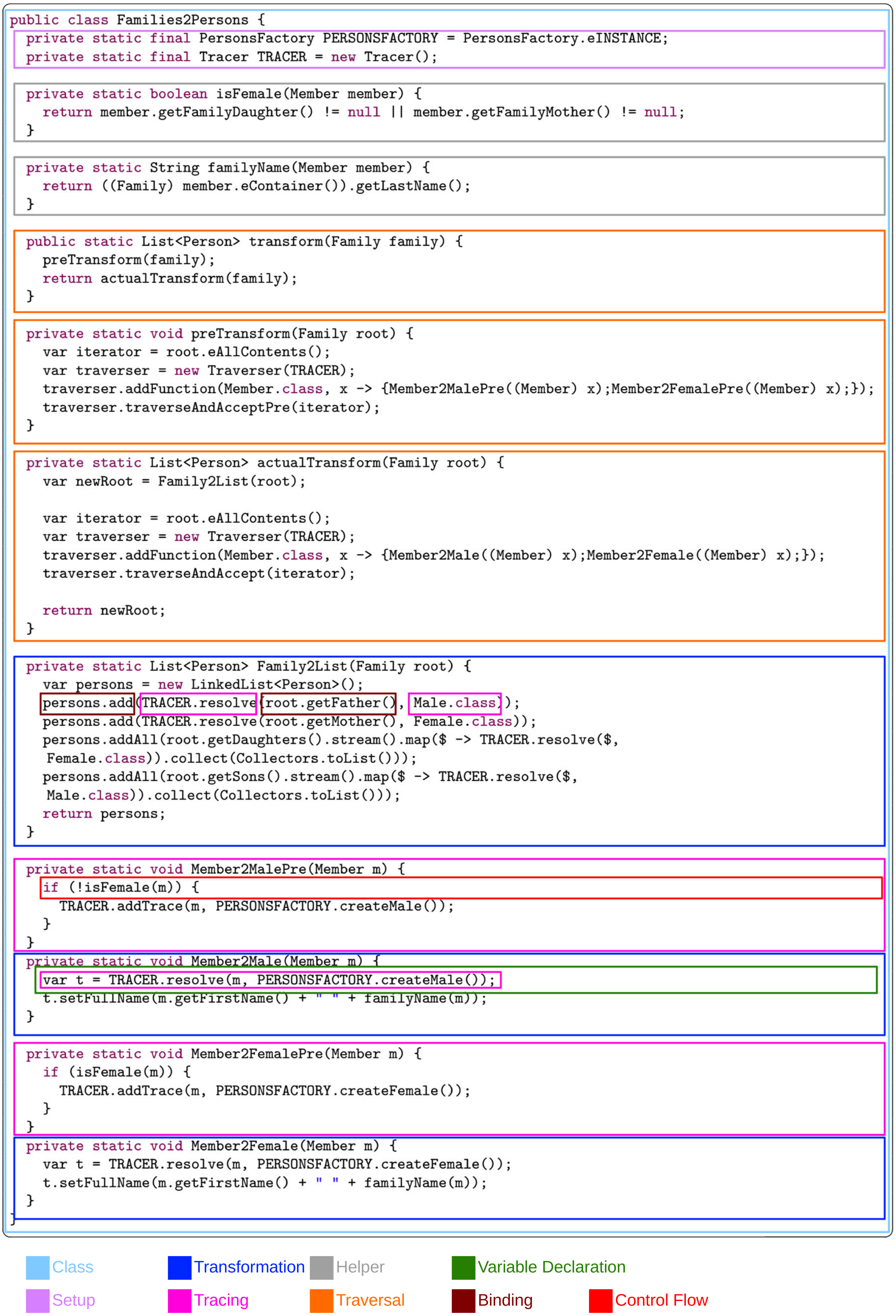
Lastly, we can also label different parts of a single line with different labels based on their functionality. Line *38* of Fig. 5, for example, has elements that perform assignments, i.e. bindings translated to Java, and elements that perform additional tracing operations. The labelling of this line reflects these different functionalities within the line by labelling substatements within the line instead of the whole line.

5 Size and complexity analysis methodology

Our analysis of the transformation specifications is guided by the research questions introduced in Sect. 1.2.

5.1 RQ1: How much can the complexity and size of transformations written in Java SE14 be improved compared to Java SE5?

To compare the transformations written in Java SE14 and Java SE5, we decided to use code measures focused on code complexity and size. For this reason, we chose McCabe's cyclomatic complexity and LOC which are shown to correlate with the complexity and size of software [29]. To keep the LOC count as fair as possible, all Java code was developed by the same researcher and we used the same standard code formatter for all Java code. Furthermore, we supplement LOC with an additional measure for code size based on word count, the combination of these two measures also allowed additional insights. Word count means the number of words that are separated either by whitespaces or other delimiters used in the languages, such as a dot (`.`) and different kinds of parentheses (`()` `[]` `{}`). This measure supplements LOC because it is less influenced by code style and independent from keyword and method name size [18]. This method for calculating transformation code size has already been successfully used by Anjorin, Buchmann, Westfechtel, et al. [18] to compare several (bidirectional) transformation



```java
public class Families2Persons {
  private static final PersonsFactory PERSONSFACTORY = PersonsFactory.eINSTANCE;
  private static final Tracer TRACER = new Tracer();

  private static boolean isFemale(Member member) {
    return member.getFamilyDaughter() != null || member.getFamilyMother() != null;
  }

  private static String familyName(Member member) {
    return ((Family) member.eContainer()).getLastName();
  }

  public static List<Person> transform(Family family) {
    preTransform(family);
    return actualTransform(family);
  }

  private static void preTransform(Family root) {
    var iterator = root.eAllContents();
    var traverser = new Traverser(TRACER);
    traverser.addFunction(Member.class, x -> {Member2MalePre((Member) x);Member2FemalePre((Member) x);});
    traverser.traverseAndAcceptPre(iterator);
  }

  private static List<Person> actualTransform(Family root) {
    var newRoot = Family2List(root);

    var iterator = root.eAllContents();
    var traverser = new Traverser(TRACER);
    traverser.addFunction(Member.class, x -> {Member2Male((Member) x);Member2Female((Member) x);});
    traverser.traverseAndAccept(iterator);

    return newRoot;
  }

  private static List<Person> Family2List(Family root) {
    var persons = new LinkedList<Person>();
    persons.add(TRACER.resolve(root.getFather(), Male.class));
    persons.add(TRACER.resolve(root.getMother(), Female.class));
    persons.addAll(root.getDaughters().stream().map($ -> TRACER.resolve($,
     Female.class)).collect(Collectors.toList()));
    persons.addAll(root.getSons().stream().map($ -> TRACER.resolve($,
     Male.class)).collect(Collectors.toList()));
    return persons;
  }

  private static void Member2MalePre(Member m) {
    if (!isFemale(m)) {
      TRACER.addTrace(m, PERSONSFACTORY.createMale());
    }
  }
  private static void Member2Male(Member m) {
    var t = TRACER.resolve(m, PERSONSFACTORY.createMale());
    t.setFullName(m.getFirstName() + " " + familyName(m));
  }

  private static void Member2FemalePre(Member m) {
    if (isFemale(m)) {
      TRACER.addTrace(m, PERSONSFACTORY.createFemale());
    }
  }
  private static void Member2Female(Member m) {
    var t = TRACER.resolve(m, PERSONSFACTORY.createFemale());
    t.setFullName(m.getFirstName() + " " + familyName(m));
  }
}
```

Fig. 5 Partially labelled Java solution for the Families2Persons case

languages including eMoflon [30], JTL [31], NMF Synchronizations [32] and their own language BXtend [33]. Their argument for using word count is that because it approximates the number of lexical units it more accurately measures the size of a solution than lines of code.

We applied the Java code metrics calculator (CK) [34] on all 24 transformations (12 Java SE5 + 12 Java SE14) to calculate both metrics and used a program developed by us to calculate the word count measure. For a basic overview we then compare the total size between Java SE5 and Java SE14 based on both LOC and word count and discuss observations as well as possible discrepancies between the two measures. The same is done for McCabe complexity as well. Because CK calculates metrics on the level of *classes*, *methods*, *fields* and *variables* we opted to additionally use the values calculated on the level of *methods*, i.e. the LOC, word count and McCabe complexity of the method bodies, to gain a more detailed understanding of where differences in size and complexity arise from. Since neither the *fields* level nor the *variables* level contained values for McCabe complexity and no interesting values for LOC and word count we decided to omit data from those in our analysis. The metric values calculated by CK were then analysed and compared based on maximum, minimum, median, and average values.

RQ1 serves the purpose of providing a general overview of the differences between the code size and complexity between Java SE5 and Java SE14. The results from this research question are analysed and discussed in more detail in **RQ2&3**.

5.2 RQ2: How is the complexity of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to each other and ATL?

To answer **RQ2**, we compare the distribution of complexity within the Java code with regard 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. To be able to analyse the complexity distribution in Java transformations, it is necessary to differentiate the different steps within the Java code, i.e. *model traversal*, *transformation*, *tracing*, *setup* and *helper*. Since cyclomatic complexity can not be calculated for each line but only for set of instructions we decided to fall back on the granularity of methods and use the classification and labelling given to each method in Sect. 4.

Based on the classification introduced in Sect. 4.2, all Java transformations were labelled by one author. The labelling was verified by the other two authors with one of them cross-checking 2 transformations and the other one checking 4. The checked transformations were *istar2archi*, *Palladio2UML*, and *R2ML2XML* all in both Java SE5 and Java SE14 which in total meant that about 51% of the total Java code lines were reviewed.

We then used the measures calculated for **RQ1** to create plots of the complexity distribution. The distribution shown in the resulting plots was then analysed taking into account the results of Götz and Tichy [16] regarding the distribution of different transformation aspects in ATL. The goal in this step was to see how the complexity in Java transformations is distributed onto transformation aspects, such as tracing and input model traversal, that are abstracted or hidden away in ATL as well as to see the evolution of this distribution between the two different Java versions.

5.3 RQ3: How is the size of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to each other and ATL?

The approach for this research question is twofold and follows a top down methodology. First, we compare the distribution of code size within the Java code over the different transformation aspects using the classification from Sect. 4. 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.

We opted to use word count as a measure for the detailed discussion of code size. The reason why we use word count and not lines of code lies in their granularity. For some parts of our analysis, it is necessary to split the value of single statements up into that of their components. This is much easier to do when using word count as a measure and does not require code to be rewritten in an unintuitive way. Moreover, the finer granularity also allows a more detailed look into the structure of methods that was not possible in **RQ2** due to the limitation of cyclomatic complexity.

The idea behind our approach is to calculate the word count for all transformations written in Java and ATL and then compare both the total count of words as well as the number of words required for specific aspects within the transformation process. While the word count for Java transformations is calculated specifically for this study, the data for the ATL transformations are taken from the results of Götz and Tichy [16].

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

List. 15 A rule with a simple binding.

Based on the introduced categorizations, we then create Sankey diagrams for the distribution of word count in both

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

List. 16 A rule with a binding using traces.

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

List. 17 A typical helper in ATL.

Java and ATL. These graphs then form the basis for our comparison. Here, we compare both the distributions of the individual transformation aspects in Java with ATL as well as the concrete sizes on the basis of the numbers. When comparing the size distribution, we analyse how the distribution of the transformation aspects in Java differs from ATL, i.e. which aspects are disproportionally large or small compared to ATL. We also explicitly look at how much code is required for tracing in Java. For this, we look at the proportion of the transformations that require traces and how that compares to the total size of Java code related to traces. Lastly, the total number of words between Java and ATL are also directly compared to see which language allows for shorter transformation code based on this measure.

To illustrate where the observed effects originate from, we use a selection of three ATL fragments representing code which is often written in ATL transformations. The first fragment (see Listing 15) 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 [16]. The second fragment (see Listing 16) represents code that requires ATL to use traceability links, which [16] 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 cannot 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 17) is a helper definition of average 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.

5.4 RQ4: How does the size of query aspects of transformations written in Java SE5 & SE14 compare to each other and ATL?

As previously discussed, the goal of this research question is to investigate the claim that writing queries for models was improved with the introduction of model transformation languages such as ATL and to check if this is still the case when utilizing new languages features in general purpose languages today. This discussion of Java vs OCL has already been raised approaches to replace OCL with Java [35]. The data basis for this analysis is formed by all helpers and their corresponding Java translations in form of methods within the 12 transformation modules subject in this study. Because this set only contains a total of 15 helpers, we complement it with a large collection of helpers and their translations from a set of supplemental libraries used in the *UML2Measure* transformation.

In our analysis, we compare Java and ATL helpers first based on their total word count and then by contrasting each ATL helper with its Java counterpart using regression analysis. All observations in this analysis are supplemented with code segments that highlight them. The regression analysis uses a linear regression model to predict the word count of Java methods ($J5WC$, $J14WC$) based on the word count of ATL Helpers ($HelperWC$). This was chosen based on an hypothesis that Java code entails an additional fix cost compared to OCL expressions as well as an increase by some factor due to the more verbose syntax of Java. This approach allows us to both verify the hypothesis and identify an approximation of the interrelationship between the code sizes.

6 Results

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

6.1 RQ1: How much can the complexity and size of transformations written in Java SE14 be improved compared to Java SE5?

Table 3 presents an overview of lines of code (LoC), word count (# words) and the sum of McCabe complexities of all methods contained in the transformation classes (WMC). Looking at the total lines of code and WMC, the numbers display an expected decrease in both size and complexity. Our transformations written in Java SE5 total 3252 lines of code and have a WMC of 792. The same transformations written in Java SE14 require only 2425 lines of code and have a WMC of 411. Based on these measures, the size reduces by about 25%, while the cyclomatic complexity is cut in half to about 52% of its Java SE5 counterpart. This decreased WMC can be attributed to the improvements made through utilizing streams for handling collections. The traversal library also contributes to this by removing all control flow branching for the `transform` methods and thus reducing the McCabe complexity of these methods.

The word count measure, however, shows a different picture. While the Java SE5 implementation uses 13007 words, the Java SE14 implementations use nearly the same amount of words, 13118 to be exact. When combining this with the reduced number of code lines provides and interesting observation. Transformation code written in Java SE14 for our transformation set is more dense, i.e. a single line of code contains a lot more words and thus more information about the transformation.

Overall, both the total number of lines of code as well as the WMC of transformations in the newer Java version are greatly reduced. However, there is no notable change in the number of required words, which hints at a more information-dense code rather than simply less code.

Table 4 summarises the calculated size (LOC and word count) and complexity (McCabe) measurements on the method level for both the Java SE5 and Java SE14 transformation code.

As expected from the total numbers, the average and median length, measured in LoC, of methods in Java SE14 is reduced by about 30%. The already low minimum of 3 lines has not been further reduced in the newer version, but the longest method is now 51 lines shorter.

Contrasting the numbers for lines of code with word count, we see a small increase in both the average and median method sizes in Java SE14 compared to Java SE5. However, the maximum number of words for a method is about 43% shorter in Java SE14 than in Java SE5. This means that while on average (or median) the number of words required to implement transformation-related methods in Java SE14 increased compared to Java SE5, newer Java versions help to reduce the size of methods that required large number of words in older Java versions.

The reduction in cyclomatic complexity seen in the total numbers is also reflected for the more detailed consideration on method level. The average transformations written in Java SE14 are 45% less complex than in Java SE5. A result also reflected in the median. Furthermore, the maximum McCabe complexity is reduced from 44 to 11, which is a significant decrease as this suggests that even highly complex methods within the transformations can be expressed a lot less complex in newer Java versions. This, again, can be attributed to the utilization of streams and functional interfaces which help to remove the requirement to manually implement large amounts of loops and nested conditions.

The more detailed results reflect what was already shown on a coarse-grained level. Compared to Java SE5, new language features in Java SE14 help to reduce the required number of code lines, while the number of words stays about the same. The cyclomatic complexity is significantly reduced, most prominently seen in the fact that the most complex method in Java SE14 is only 1/4th of the complexity of the most complex method in Java SE5.

6.2 RQ2: How is the complexity of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to ATL?

The results for this research question are split up into two parts. We first report on our findings for Java SE5 and its comparison to ATL in Sect. 6.2.1, before reporting the findings for Java SE14 and its comparison to ATL and Java SE5 in Sect. 6.2.2.

6.2.1 Java SE5

Figure 6 shows a plot over the distribution of WMC split up into the different transformation aspects involved in a transformation written in Java SE5 and Java SE14. It shows that about 60% of the complexity involved in writing a transformation in Java SE5 stems from the actual code representing the transformations and helpers. The other 40% are distributed among the model traversal, tracing, and setup code. In ATL, these three aspects are completely hidden behind ATL's syntax. In other words, this means that 40% of the complexity within the transformations written in Java SE5 stems from overhead code.

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.

6.2.2 Java SE14

Given the observations from **RQ1** combined with 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 in Fig. 6, 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 in Java SE14.

While the complexity associated with model traversal is greatly reduced by the use of the traversal library, the overall

Table 3 Measurement data on the translated transformation modules

Transformation Name	LOC		# words		WMC	
	Java SE5	Java SE14	Java SE5	Java SE14	Java SE5	Java SE14
ATL2BindingDebugger	22	19	93	88	4	2
ATL2Tracer	74	17	285	283	7	5
DDSM2TOSCA	509	339	2137	2036	103	44
ExtendedPN2ClassicalPN	147	107	569	553	37	19
Families2Persons	72	62	273	297	22	14
istart2archi	184	115	689	714	57	24
Modelodatos2FormHTML	215	178	761	750	58	40
Palladio2UML	303	253	1066	1100	70	47
R2ML2XML	1181	855	4720	4966	303	139
ResourcePN2ResourceM	99	67	380	389	29	13
SimpleClass2RDBMS	163	111	629	581	50	26
UML22Measure	283	249	1405	1356	52	38
Total	3252	2425	13007	13118	792	411
Median	173.5	113	599	647	51	25
Average	271	202.1	1088.9	1092.75	66	34.25

Table 4 Measurement data on the methods in the translated transformation modules

Measure	Minimum		Median		Average		Maximum	
	Java SE5	Java SE14	Java SE5	Java SE14	Java SE5	Java SE14	Java SE5	Java SE14
LoC	3	3	7	6	12.5	9.4	135	105
# words	1	2	5	6	5.2	6.4	64	37
McCabe complexity	1	1	2	1	3	1.6	44	11

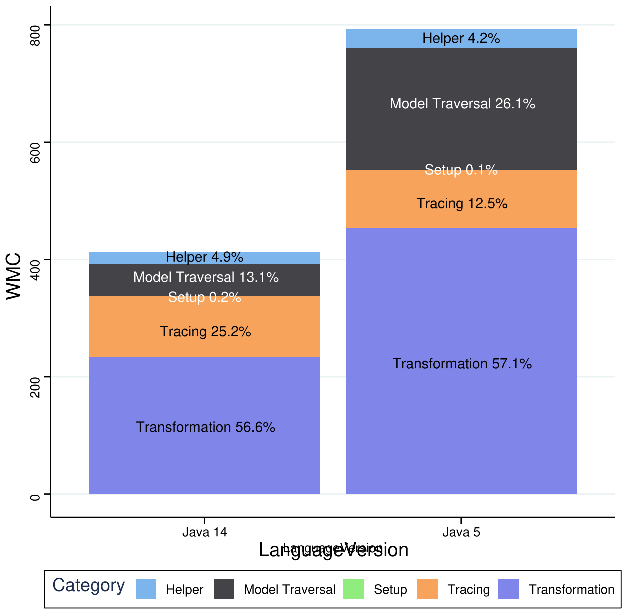
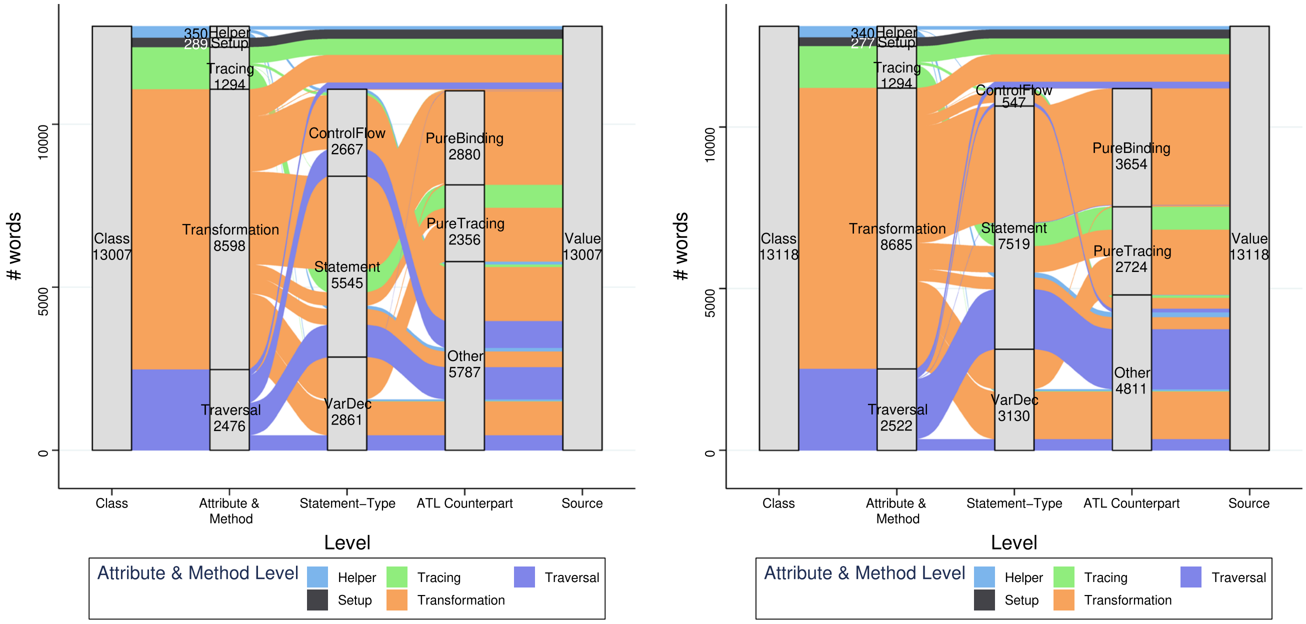


Fig. 6 Distribution of WMC over transformation aspects in Java SE5 and SE14

distribution between the actual code representing the transformations and helpers and the model traversal, tracing, and setup code does not change much. About 40% of the overall transformation specification complexity still stems from 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. 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 methods 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, handling traces still entails a large overhead.



(a) Distribution in Java SE5.

(b) Distribution in Java SE14.

Fig. 7 Distribution of word count over transformation aspects in Java SE5 and SE14

6.3 RQ3: How is the size of transformations written in Java SE5 & SE14 distributed over the different aspects of the transformation process compared to ATL?

The reporting of results for this research question follows the same structure as Sect. 6.2. First in Sect. 6.3.1 the results of our analysis of Java SE5 and its comparison with ATL are reported. Afterwards in Sect. 6.3.2 the results for Java SE14 and its comparison with ATL are discussed. This section also contains a comparison to the results of Java SE5.

6.3.1 Java SE5

The total size of Java SE5 transformations compared to ATL transformations is much larger when using word count as a measure. All ATL transformations in our set together amount to 7890 words, while the Java SE5 code needs 13007. This is an increase of 64.8%. Figure 7a allows us to look at the distribution of written words over the transformation aspects introduced in Sect. 4. The x-axis of the graph describes the hierarchy levels from Sect. 4. The word count is depicted on the y-axis, and on each hierarchy level on the x-axis the word count distribution of its different aspects is shown. How each level is made up of its sub-levels is then shown by means of the alluvial lines flowing from left to right. The flow lines are coloured according to the Attribute & Method level as it represents the top level of separation and eases readability.

Looking at the graph we see a large portion of the number of words is actually associated with the transformation code itself. Overhead from tracing, traversal, and setup exists, but it is not as prevalent as expected from the results presented in Sect. 6.3. However looking more closely into each of the aspects and their makeup reveals that there is more overhead still hidden in the transformation-related code. In the following, we will look at the individual aspects and their more precise breakdown and what this means for transformations written in Java SE5, also in comparison with ATL.

The number of words required to express Helper code for our transformation set is low. It constitutes 2.9% of all words within the transformation class which is in line with the size of helpers in ATL as seen in Fig. 8.

Similarly, the number of words required for setup code is also of little consequence as it constitutes only about 2.2% of the total word count in the transformations considered in this work. However, even though the amount is small, the code still has to be written and maintained when evolving the transformation.

Another part of the code within the transformation classes that represents overhead in Java SE5 compared to ATL is the code related to tracing. While ATL abstracts away tracing and does target element creation implicitly, in Java this behaviour has to be recreated by hand. The library for tracing introduced in Sect. 3.3 helps reduce the implied overhead, but the creation of target objects as well as traces for them still has to be initiated manually. The methods involved in this constitute for 9.9% of words used in our translated transformations and

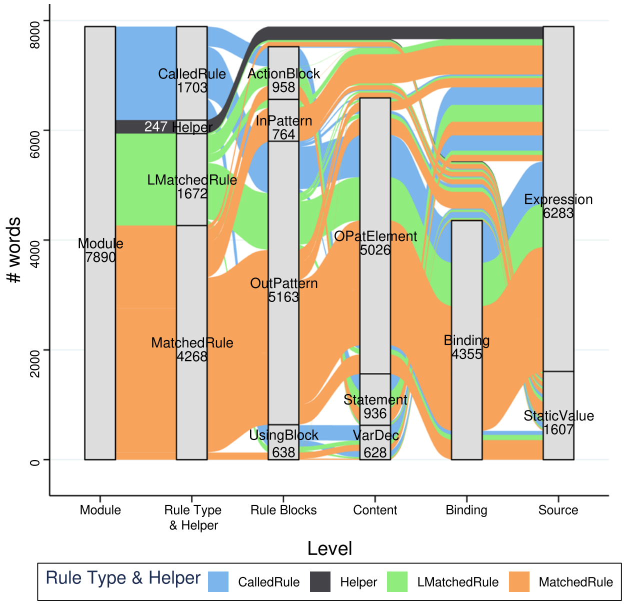


Fig. 8 Distribution of word count "complexity" measure over transformation aspects in ATL calculated based on Götz and Tichy[16]

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

List. 18 A rule with a simple binding in Java SE5.

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

List. 19 A rule with a binding using traces in Java SE5.

are made up of methods in style of what is described in Sect. 3.4.

As previously stated, a large portion (65.8%) of the word count comes from methods and attributes related to the actual transformation. This however changes when looking at the lower levels of classification within those methods. In ATL 60% of the total number of words and 61% of the words within rules stem from bindings, i.e. the core part responsible for transforming input into output. In our Java SE5 translation this differs greatly. The translated binding code only makes up 22% of the total word count or 33.5% within the transformation methods. This points to the fact that much less of what is written in Java SE5 actually relates to actual transformation activities. In Java many more words are spent on code not directly transformation-related but rather on tasks necessary for the transformation to work. Three such types of code stand out.

One is statements that resolve traces built up in the tracing methods discussed in the last section (as seen by the flow from Transformation over Statement and Variable Declarations towards Tracing in Fig. 7a). Examples of such code in the Families2Persons example from Fig. 5 and Listing 8 can be found in lines *38*,*39* and *51*.

The second one is code to initialise temporary variables used for processing steps within the transformation (as seen by the flow from Transformation over Variable Declarations into Other in Figure 7a).

And lastly there are a large number of words associated with control flow via loops and conditions to process collections in order to bind their transformed contents onto attributes of the current output object (as seen by the flow from Transformation over Control Flow towards Other in Fig. 7a).

Code relating to traversal is again overhead introduced due to the usage of Java over ATL. The number of words required for writing traversal-related code for our set of transformation constitutes 18.9% of the total word count of transformation classes.

Overall, the overhead produced by Tracing, Traversal and Setup code amounts to 31% of the total number of words for our Java SE5 transformations. Furthermore, while 65.8% of words within the transformation classes are related to the process of transformation, many of them are again overhead from manual trace resolving, model traversal and supplemental code.

When comparing a simple binding (see Listing 15) written in ATL with its translation in Java SE5 (see Listing 18), there is not much difference. Both require nothing more than their language constructs for accessing attribute values and assigning them to a different attribute.

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 16), some explicit code needs to be written in Java (see Listing 19). As a result, the transformation specification gets larger 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 Sect. 3.3.3, the trace library holds `EObjects` which have to be converted to the correct type after they have been retrieved based on the source object.

The increase in size is even more prevalent when looking at the translation of a typical helper. The helper in Listing 17 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 20, the code gets a

```
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 }
```

List. 20 A typical helper in Java SE5.

lot more complex and bloated. This is due to, as previously stated in Sect. 6.3, 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). We investigate and discuss this in more detail later in Sect. 6.4.

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.

6.3.2 Java SE14

Comparing the total number of words in Java SE14 transformations with ATL, a similar picture as for Java SE5 arises. The translated transformations require 13118 words, while ATL only requires 7890. Surprisingly, as also discussed in Sect. 6.1, the number of words in Java SE14 is higher than that of Java SE5, although only by around 100 words, despite requiring less lines of code and cyclotomic complexity. We believe this to be the result of two effects. One, using streams for processing collections reduces the lines of code and cyclomatic complexity because they are single statements and are thus not split over as many lines as when using loops. But, setting up streams and transforming them back into the original collection requires several additional method calls which offset the overall reduction of number of words.

The distribution of the number of words between Java SE5 and Java SE14 also differs immensely, especially around the makeup of transformation methods, as evident from Fig. 7b. It also again highlights key differences between the ATL transformations and their Java counterparts.

```
for (InElement i : input.getInElements()) {
    output.getOutElements()
     .add(TRACER.resolve(i, OutElement.class));
}
```

List. 21 Trace resolution example of a collection in Java SE5.

The portion of words required for writing Setup and Helper code has slightly reduced compared to Java SE5, while the proportion of words for Transformation and Traversal methods increased. The Methods & Attributes for setting up helpers does not change which is due to the fact that the underlying code does not change between Java SE5 and Java SE14.

Thus, more can be concluded from how the number of words are distributed within the Transformation and Traversal methods in Java SE14.

For Traversal, it is noticeable that almost no control flow statements are used any more. Instead, most words now come from simple statements. This is because in Java SE14 we make use of the Traversal library, which allows us to pass only the classes to be matched and the methods to be called to the traverser instead of having to write loops and conditions manually. This evidently does not reduce the number of words, but it creates a different way of defining traversal.

Similarly, the transformation-related methods in Java SE14 also contain much less words that define control flow. The number of words for other statements not directly performing transformation tasks is also reduced. Instead, the translated bindings now make up a larger proportion of the word count. In our Java SE14 transformations, the code for translated bindings now makes up 27.8% of all words compared to the 22% in Java SE5 and 41.9% of words within the transformation methods. This stems from the usage of streams for processing collections of input elements rather than explicit loops and conditions. As a result the Java SE14 implementation is less control flow driven and focuses more on the data involved. However, while this allows for less lines of code and a reduction in cyclomatic complexity as shown in Sect. 6.1, it does not improve the required number of words. This is because in some cases, the setup overhead for streams counteracts their conciseness gain when using number of words as a measure. An example of this can be seen when comparing Listings 21 and 22. Both code segments resolve all `InElements` from the input into their corresponding `OutElements` and add them to the `OutElements` list of the output. The number of words required in Java SE5 for this totals 14, whereas the number of words in Java SE14 amounts to 17.

```
output.getOutElements()
 .addAll(input.getInElements().stream()
   .map(i -> TRACER.resolve(i, OutElement.class))
   .collect(Collectors.toList()));
```

List. 22 Trace resolution example of a collection in Java SE14.

Overall, our translated transformations in Java SE14 do not reduce the number of words compared to their Java SE5 counterpart. Newer language features do however help in reducing the amount of explicit control flow statements and supplemental code required. Most of this is now done directly in translated bindings which more closely follows the ATL-style. In this sense, Java SE14 helps to take a more data-oriented approach to transformation development compared to Java SE5. However, there is still much overhead from manual traversal, tracing and supplemental code compared to ATL.

When comparing the code segments 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.

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

List. 23 A typical helper in Java SE14.

Comparing translated helper code, however, does show some improvements of Java SE14 over Java SE5. Because of the introduction of the streams API, Java SE14 (see Listing 23) can now handle expressions involving collections nearly as seamless as ATL (see Listing 17). Only the overhead of calling `stream()` and `.collect(Collectors.toList())` remains. This and other observations regarding OCL expressions translated to Java are discussed in more detail later in Sect. 6.4.

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.

6.4 RQ4: How does the size of query aspects of transformations written in Java SE5 & SE14 compare to each other and ATL?

Comparing the word count numbers of helpers from the transformation modules and libraries with their translated counterparts we can once again observe an increase in Java. While all helpers in ATL combined total 2299 words the Java SE5 code totals 3801 words which is an increase of about 65.3%. This was to be expected since Java SE5 is more verbose, especially when handling collections which are required for all helpers within the libraries. This becomes clear when looking at the Java SE5 translation of Listing 17 in Listing 20. Not only does Java require a loop and if-condition to filter out the desired association subset, a new results list also has to be created and filled with values. Compared to OCLs "→ syntax" this increases the number of required words to produce the same result drastically.

Next, as described in Sect. 5.4 a linear regression was calculated to predict the word count of Java SE5 code for Helpers based on their word count. We were able to find a significant regression model ($p < 2.2e-16$) with an adjusted R^2 of 0.649. The predicted word count of Java SE5 expressions for OCL expressions is estimated as $4.85364 + 1.31554 * HelperWC$. The hypothesis of a linear relationship is also supported by a Pearson coefficient of 0.81 indicating this linear relationship.

Overall, we see a linear relationship between OCL expression code and the translated Java SE5 code. The factor with which the Java code increases in size more quickly is 1.53. This combined with the subjectively less clear way of handling collections through loops leads to the observation that Java5 was not well suited for defining expressions on models.

Looking at the number of words of Java SE14 Helpers compared with their ATL counterparts we see a similar but slightly smaller size than with Java SE5. As stated earlier all ATL library helpers total 2299 words and with 3350 words their Java SE14 counterpart is only about 45.8% larger compared to the 65.3% of Java SE5. This fits well into our observation that the verbose handling of collections is responsible for large portions of the size increase. The streams API, introduced in Java SE8, allows developers a less verbose way of handling collections as can be seen when comparing Listings 20 and 23. While there is still some overhead compared to the OCL counterpart, namely the necessary calls to `stream()` and `.collect(Collectors.toList())`, the total overhead is greatly reduced. Moreover, this difference could in principal be eliminated by using an alternative GPL. The Scala programming language, for example, does not require a conversion between streams and collections.

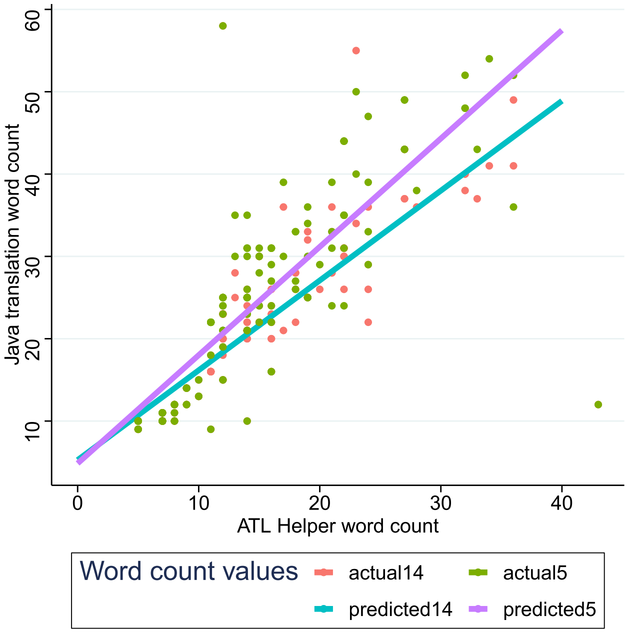


Fig. 9 Comparison of actual Java SE5 and SE14 helper size with predicted size based on linear the regression models

The decrease in size can also be observed in our linear regression model that predicts the word count of Java SE14 code for OCL expressions based on the word count of those expressions. The model we were able to find is significant ($p < 2.2e{-}16$) and has an adjusted R^2 of 0.64. The predicted word count of Java SE14 expressions for OCL expressions is estimated as $5.26631 + 1.09064 * HelperWC$. And the hypothesis of a linear relationship is again supported by a Pearson coefficient of 0.8. Figure 9 shows how well both the regression models fit the data. It also highlights the decrease of words required for translated helpers in Java SE14 compared to Java SE5.

The x-axis depicts the word count value of OCL expressions, while the y-axis depicts the word count of Java SE5 codes. The dots within the graph then show the corresponding Java SE5 code word count for each translated OCL expression. Lastly, the red line shows the predicted correspondence based on our regression model.

Overall, we still see a linear relationship between OCL expression code and the translated Java SE14 code. However, the factor with which the Java code increases in size more quickly is only approximately 1.1. This leads us to believe that a well trained Java developer should be able to express OCL queries in Java without much difficulties.

7 Discussion

In this section we discuss our findings from Sect. 6 as well as our experiences from the process of translating and using transformations in Java. Our discussion revolves around two main topics. First, we want to discuss the impact that the design decision to not use anonymous classes to outsource traversal in our Java SE5 solution, explained in Sect. 3.3.2, has on the presented data. Then we discuss how the advancements that have been achieved in newer Java versions influence the ability for developers to efficiently develop transformations in Java. This also includes a conversation about what shortcomings still exist. And second we present a guide that suggests in what cases general purpose languages such as Java can be used in place of ATL. We also show cases where we would advise against writing transformations in Java because of its disadvantages. The argumentation of this part is based on the results presented in this publication as well as our experiences, both from this study as well as previous works [8–12]. Finally, we want to have a short discussion beyond the results of our study. Here we want to talk about other features that MTLs can provide and what those could mean for the comparison of MTLs vs. GPLs.

7.1 The impact of not outsourcing model traversal in Java SE 5

As explained in Sect. 3.3.2, we decided on using the conditional dispatcher pattern to implement traversal in our Java SE5 solution as opposed to implementing a traversal library, similar to the one used in Java SE14, using anonymous classes. This design decision has implications for the data presented throughout Sect. 6 which we discuss here.

As mentioned, using the presented approach leads to an increased McCabe complexity for the traversal implementation in Java SE5, while it reduces the LOC and number of words. This has concrete implications for the numbers discussed in Sections 6.1 to 6.3.

For one, this means that when comparing the concrete numbers as done in Sect. 6.1, the stagnation of number of words observed between the Java SE 5 and Java SE 14 variants, would not be present with the alternative Java SE 5 implementation. This is because it would be 812 words longer (making the total number of words 13819) than the presented implementation and thus one would instead observe the expected decrease in number of words in the Java SE 14 implementation. It would still not be as significant, because only the traversal part of all transformations are affected, but it would be more in line with the reduction in code size observed with the LOC measure in the presented implementations. Moreover, the LOC reduction itself would also be more pronounced because the alternative traversal implementation does require more lines of code per rule.

Specifically the total of the Java SE 5 implementation would be increased by 1020 LOC to a total of 4272 as opposed to 3252.

The difference in WMC between the Java SE 5 and Java SE 14 implementation on the other hand would be less clear-cut. As shown in Fig. 6 a significant portion of the WMC in the presented Java SE5 implementation stems from model traversal. In the alternative implementation this complexity would be significantly reduced by 152 to a total of 640 as opposed to 792. The overall WMC of the Java SE 5 transformations would still be higher, because the utilisation of streams in Java SE 14 reduces the McCabe complexity of other parts of the transformation as well, but it would no longer be nearly halved.

Our observations regarding the differences between implementations in the two different Java versions would, however, not change significantly with the alternative Java SE 5 implementation. Thanks to the functional interfaces and streams, in newer Java versions, a more declarative style for defining transformations can still utilised. The WMC of the code is also still reduced, and the general focus can be directed a more towards the actual transformation aspects. In addition, the observations regarding the comparison of Java and ATL do not change.

7.2 Language advancements and their influence on the ability to write transformations: a historical perspective

The overall number of words required to write transformations in Java SE14 compared to Java SE5 has not reduced, as shown in Sections 6.1 and 6.3. However, we have also seen that less explicit control flow needs to be written and the focus shifts more to the binding expressions. This shows in the results discussed in Sections 6.1 and 6.2 as the cyclomatic complexity of transformations written in Java SE14 is greatly reduced. In principle, a shift towards more data-driven development of transformations is therefore possible. Whether this brings an overall advantage or not is still a debated topic [2] and in our eyes depends on the experience and preference of the developers. However, there are many studies in the field of object-oriented programming that establish a connection between cyclomatic complexity and reliability [36–40], i.e. fault-proneness and error rate, as well as some that establish a connection between cyclomatic complexity and maintainability [41,42], i.e. change frequency and change size.

It has been our experience that newer Java features such as streams and the functional interfaces make the development process easier because less work has to be put into building the traversal, and the assignments within the transformation methods are now a more prominent part of them, i.e. they are less hidden in loops and conditions. Whether these advancements justify writing transformations in Java compared to ATL is discussed in the next section.

7.3 A guideline for when and when not to use Java or similar GPLs

As shown in Sections 6.2 and 6.3, while newer Java features shift the focus more towards a transformation-centric development, there is still significant overhead from setup, manual traversal, and especially tracing. Of those three, we believe the setup overhead to be of least relevance. That is because the total overhead for setup is small and it is only an initial overhead that, for the most part, does not need to be maintained throughout the lifecycle of a transformation. The situation is similar for traversal overhead. The code required to be added for all rules or transformation methods, while more significant in its size still only needs to be written once and can be ignored for most of the remaining development. There is little to no room for errors to be introduced , in any Java implementation that follows a style similar to our implementations, as each new rule requires nearly identical code to be added.

Tracing is where, in our opinion, most of the difficult overhead arises from. It is thus the main argument for writing transformations in ATL or similar MTLs compared to general purpose languages. Managing traces and implementing their complete semantics cannot be outsourced into a library, but we can only use a library to reduce the required effort. For many of the advanced use cases, the mapping semantic relies on String constants that are passed to both the creation and resolution methods, which is error-prone. Such cases arise when traces to objects are needed that were only a side effect of a transformation rule and not its primary output.

There is also little support through type-checking since the only way to store traces for all elements is to use the most generic type possible (i.e. `EObject`). This results in the burden of creating and fetching objects of the correct type to be shifted to the developer , which constitutes a clear disadvantage compared to ATL, where trace resolution is type-safe. In simple cases, this problem is less conspicuous, but in cases where advanced tracing is required, much of the described difficulties arise and can lead to errors that are hard to track to its origin. It also forces developers to be more aware of all parts of the transformation at all times, to make sure not to miss any possible object types that could be returned from resolving a trace. There are approaches, such as Goldschmidt, et al. [43], that bring type safety to GPL transformations, but they also come with their own set of limitations when considering advanced features such as incrementality and reusability of the introduced templates, that developers need to be aware of, as well as other boilerplate code that is required to set it up.

Based on the presented reflection, we believe that general purpose languages largely excel in transformations where little to no tracing and especially no advanced tracing is required. The overhead for setup and traversal is manageable in these cases. Moreover, when no traces are required for the transformation, we can scrap the two-phase mechanism completely and thus half the total overhead of traversal is required.

There is also an argument to be made about the expressiveness of Java for complex algorithms compared to the limited capabilities of OCL. We were faced with such a concrete case during the development of a model differencing tool called SiLift [8]. SiLift takes a so-called difference model as input and aims at lifting the given input to a higher level of abstraction by applying in-place transformations to group together interrelated changes. To achieve this low-level changes comprised by the given difference model are first grouped to so-called semantic change sets in a greedy fashion. This greedy strategy, however, can lead to too many change sets. Specifically, we need to get rid of overlapping change sets in a second phase of the transformation, referred to as post-processing in Kelter, and Taentzer [44]. The post-processing poses a set partitioning problem which may be framed as an optimization problem: We want to cover all low-level changes by a minimum amount of semantic change sets which are mutually disjoint. We implemented the heuristics presented in Kehrer, Kelter, and Taentzer [44] in Java. This can be hardly expressed in OCL, which was developed as a language for querying object structures but not for implementing complex algorithms like the post-processing step of the in-place model transformation scenario described above.

Lastly, related to the previously discussed point of expressiveness, the heterogeneity of Java code compared to ATL code also sticks out. The structure of ATL rules, enforced by ATL's strict syntax, allows for writing consistent code across different transformations. This means that developers can quickly see the basic intent of a rule. The same cannot be said for Java methods. While our translation scheme, combined with the developed libraries, produces an internal DSL for transformations, Java code is far less homogeneous due to the absence of any dedicated structure within methods that perform transformations. This can also be seen in our classification from Sect. 4.2. Each Java statement can either have transformation-specific semantics (i.e. *Binding* or *Tracing*) or perform any *other* transformation-unrelated task. This problem of intermixed transformation and non-transformation code within GPLs also persists throughout other internal transformation DSLs such as the NMF transformation languages [45], YAMTL [46], RubyTL [47], or SiTra [48]. But this does not only bring disadvantages. The strict structure of ATL allows to easily design mappings from one input type to one output type. This can suffice in many cases as highlighted by Götz and Tichy [16]. However, in cases where several different input types need to be matched to the same output type (n-to-1), one input type needs to be matched to several output types (1-to-n), or a combination of the two cases (n-to-m), code duplicates are often unavoidable. In heterogeneous Java code, such situations can be handled more easily. All in all, the relationship between the input and output meta-models should also be considered when deciding between using an MTL or a GPL.

7.4 Limits of our results in the context of the research field

Up till now our discussion of MTL vs. GPL largely boiled down to the abstraction of model traversal and tracing provided by ATL. This is of course by design as our study focused on the comparison of Java and ATL. ATL being the most used model transformation language and Java being one of the most dominant programming languages of the last decade. Nonetheless, there are more model transformation-specific features that other model transformation languages provide. Depending on the situation these features could also influence the decision of using a specific model transformation language over general purpose languages.

An extension of the model traversal and matching features of ATL comes in the form of graph pattern matching in graph-based model transformation languages such as Henshin [25]. This allows transformation developers to define complex model element relationships that are automatically searched and matched by advanced matching engines. There exist some advances of trying to replicate this behaviour in general purpose languages for example FunnyQT [49] or SDMLib/Fujaba [50], but even in those cases DSLs are used for defining the graph patterns.

Some model transformation languages allow to run analysis on the written transformations such as critical pair analysis [51] or even verify property preservation by a transformation [52], both of which are not easily accessible for transformations written in general purpose languages. The better analysability of MTLs stems from their syntax being transformation-specific, as also seen in the structure of our classification schemata from Sect. 4.

Being able to design bidirectional transformations based on only one transformation script is also a unique property of model transformation languages. Examples of such languages are detailed and compared in Anjorin, Buchmann, Westfechtel, et al. [18] or [53]. Some languages like eMoflon Leblebici et al.[30], NMF Synchronizations [45], or Viatra [54] extend this further by providing the ability to perform incremental transformations both being features that are hard to reproduce in general purpose languages in our experience. Even ATL now has several extensions allowing it to run incremental transformations [55,56].

Currently, for general purpose languages to be considered for writing transformations, all the stated advanced features such as graph pattern matching, bidirectional and incremental transformations as well as transformation analysis and verification should not be an essential requirement of the development. This is because none of them can be implemented with justifiable effort in GPLs.

8 Threats to validity

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

8.1 Internal validity

The manual steps done throughout our study pose some threat to the internal validity of our study. Both the translation based on our translation schema and the labelling of the Java code were done manually and thus open the possibility of human error. Furthermore the program we developed to calculate the word count of the Java code could also contain errors. We counteracted these threats by testing the correctness of the resulting transformations to the extent that was possible based on available resources. This was done by testing the output of the translated transformations against the output of the ATL transformations from which they originated as well as through rigorous peer reviews. We further verified the correctness of our labels and the produced word counts through reviews as detailed in Sect. 5.

All assumptions we make about cause and effect of increase or decrease of size and complexity as well as of overhead is supported by more detailed investigations and analysis throughout our research.

8.2 External validity

To mitigate a potential threat to the external validity of our work due to a bias in the selected transformation modules we chose the analysed transformations from a variety of sources and different authors. Moreover, both the purpose and involved meta-models differ between each transformation module, thus providing a diverse sample set.

However, the transformations chosen for evaluation in our work were subject to a number of constraints which poses a threat to the generalizability of our results. While we aimed to select a variety of transformation modules w.r.t. scope and size, the limitation of LOC may introduce a threat to the external validity of our work.

Due to the study setup of selecting ATL transformations and translating those into Java, there is the possibility of a bias in favour of ATL. It is potentially more likely for an ATL solution to exist, if the problem it solves is well suited for being developed in ATL. As a result the results of our study might not be applicable to all model transformations. However, our study does not try to confirm that ATL is the superior language for developing transformations, but discusses based on the presented observations, which advantages a dedicated language like ATL can offer. In order to be able to recognise why ATL is a good solution for certain cases, it is necessary to look at precisely such cases. In order to validate our results, a further study should be carried out. There, the study design should be reversed so that ATL solutions are derived from existing Java solutions.

Lastly, all our observations are limited to the comparison between ATL and Java which limits their generalizability. While the observations might also hold for comparing Java or similar languages with transformation languages similar to ATL, e.g. QvT-O, they cannot be transferred to graph-based transformation languages such as Henshin or even QvT-R.

8.3 Construct 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.

The used metrics for measuring complexity and size need also be discussed. We opted to use cyclomatic complexity for measuring the complexity of Java transformations because it is one of the most widely used measures for object-oriented languages and has been shown in numerous publications to relate both to the maintainability and reliability of code [29]. Because both quality attributes are of interest in the discussion of MTLs vs. GPLs, we believe the cyclomatic complexity to be a good measure to assess the impact that overhead Java code has on the quality of transformations. Likewise lines of code are a popular measure for size in all of programming but has also been criticized due to its disregard for the difference in programming styles and formatting. To counteract this problem, all Java code was developed by the same researcher using the same standard code formatter. To further counterbalance issues with lines of code as a solitary size measure, we supplemented it with the additional measure word count that has been argued to be more accurate in measuring the size of a programmed solution [18]. In cases where their ranking differs, we then investigated the cause of the discrepancy and discussed what this means for our observations and analysis.

8.4 Conclusion validity

To ensure reproducible results, we provide all the data and tools used for our study in the supplementary materials for this work. A repetition of our approach using the provided materials will end with the same results as those presented here. However, 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.

9 Related work

To the best of our knowledge, there exists no research that relates the size and 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 [13]. They compare ATL, QVT-O, and the GPL Xtend, and they found no clear evidence for an advantage when using MTLs. In comparison with 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.

As previously described, parts of our research build upon the work presented in Götz and Tichy [16]. Here, the authors use a complexity measure for ATL proposed in the 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.

In Amstel and Brand [57] the authors use McCabe complexity to measure the complexity of ATL helpers. Among others, this is also done in Vignaga [58]. 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)[6] 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 [59,60], which could serve as another source for comparing MTLs and GPLs more widely, including tooling- and execution-related aspects.

Sanchez Cuadrado et al. [28] 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 by their code and closely resembles the structure embodied in our translation schema.

Our approach to utilise libraries and define certain restrictions on the structure of code in Java defines an internal DSL for developing transformations. There exists a large body of research into the topic of the design of internal transformation languages for several general purpose languages. It would be impossible to list them all here. For this reason, we will limit our discussion to a small selection of internal DSLs which have points of contact with our Java DSL.

The Simple Transformation Library in Java (SiTra) introduced in Akehurst et al. [48] 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.

Another JVM-based transformation DSL is presented by Boronat [46]. The language YAMTL is a declarative internal language for Xtend. In contrast to our approach, this language breaks with the imperative concepts of its host language and offers an ATL-like syntax for defining transformations.

Batory and Altoyan [61] describe Aocl, an implementation of OCLs underlying relational algebra for Java. Much like OCL, Aocl allows developers to define constraints and queries for a given model using a straightforward syntax. The authors further argue that, if expanded, Aocl could be used to write model-to-model transformations, but currently this feature does not exist. Using a MDE tool it is possible to generate a Java package that allows to use Aocl for a class diagram passed to the tool.

In Hinkel and Burger [45] the authors introduce NMF-Synchronisations, an internal DSL for C# for developing bidirectional transformations. The language is built with the intention to reuse as much of the tool support from its host language as possible. Much like our Java SE14 approach, they utilise functional language constructs added to C# to allow a more declarative way of defining transformations while still retaining the full potential of the host language.

10 Conclusion

In this work, we presented how we developed and applied a translation schema to translate ATL transformations to Java. We also described our results of analysing the complexity and size as well as their distribution over the different

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

transformation aspects. For this purpose, we used McCabe complexity, LOC, and word count to measure the size and complexity of 12 transformations translated to Java SE5 and Java SE14, respectively. Based on our findings, we then discussed improvements of Java over the years as well as how well suited these newer language iterations are for writing model transformations.

We found that new features introduced into Java since 2006 help to significantly reduce the complexity of transformations written in Java. Moreover, while they also help to reduce the size of transformations when measured in lines of code, we saw no decrease in the number of words required to write the transformations. This suggests an ability to express more information dense code in newer Java versions. 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 about the same. This observation is further supported by the analysis of code size distribution. Here, we found that while large parts of the transformation classes relate to the transformation process itself, within those parts there is still significant overhead from tracing as well as general supplemental code required for the transformations to work. We conclude that while the overall complexity is reduced with newer Java versions, the overhead entailed by using a general purpose language for writing model transformations is still present.

Our regression models for predicting Java code size based on OCL expressions suggest a linear relationship for both Java SE5 and Java SE14 with the newer Java version having a slightly lower growth factor.

Overall we find that the more recent Java version makes development of transformations easier because less work is required to set up a working transformation, and the creation of output elements and the assignment of their attributes are now a more prominent aspect within the code. From our results and experience with this and other projects, we also conclude that general purpose languages are most suitable for transformations where little to no tracing is required because the overhead associated with this transformation aspect is the most prominent one and holds the most potential for errors. However, while we do not see them as prominently used, we believe that advanced features such as property preservation verification or bidirectional and incremental transformation development cannot currently be implemented with justifiable effort in a general purpose language.

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. Such studies could also investigate several other relevant aspects. For example, how well users are aided by *tool support* or the impact of *previous knowledge* of the languages or involved models on the resulting GPL or MTL code. Moreover, the impact of language choice on transformation performance, an aspect that gets more relevant with the ever increasing size of models [62], can also be investigated with our setup. Here, we envision the use of run-time measures like execution time and memory or CPU utilization to compare MTL solutions with their GPL counterparts, to investigate the scalability of the underlying technologies.

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

Funding Open Access funding enabled and organized by Projekt DEAL.

A OCL expression translations in Java SE5

```
Collection<Type> newCollection = new
    Collection<>();
for (Type t: collection) {
    if (e) {
        newCollection.add(t);
    }
}
```

List. 24 Translation of collection->select(e) in Java SE5.

```
Collection<ResultType> newCollection = new Collection<>();
for (Type t: collection) {
  ResultType r = ...; //manipulate t in accordance with e
  newCollection.add(r);
}
```

List. 25 Translation of collection->collect(e) in Java SE5.

```
boolean includes = false;
for (Type t: collection) {
    includes |= t == x;
}
```

List. 26 Translation of collection->includes(x) in Java SE5.

```
element.getAttribute();
```

List. 27 Translation of element.attribute in Java SE5.

```
Collection<AttributeType> newCollection =
    new Collection<>();
for (Type t: collection) {
    if (e) {
    newCollection.add(t.getAttribute());
    }
}
```

List. 28 Translation of collection.attribute in Java SE5.

```
if (i > 5) {}
```

List. 29 Translation of i | i > 5 in Java SE5.

References

1. Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. IEEE Softw. (2003). https://doi.org/10.1109/MS.2003.1231150
2. Götz, S., Tichy, Matthias, Groner, R.: Claimed advantages and disadvantages of (dedicated) model transformation languages: a systematic literature review. Softw. Syst. Model. **20**(2), 469–503 (2021). https://doi.org/10.1007/s10270-020-00815-4
3. Jouault, Frédéric., et al.: ATL: a model transformation tool. Sci. Comput. Program. (2008). https://doi.org/10.1016/j.scico.2007.08.002
4. Krikava, F., Collet, P., France, R.: Manipulating models using internal domain-specific languages. In: Symposium On Applied Computing. Gyeongju, South Korea (2014). https://doi.org/10.1145/2554850.2555127
5. Gray, J., Karsai, G.: An examination of DSLs for concisely representing model traversals and transformations'. In: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (2003). https://doi.org/10.1109/HICSS.2003.1174892
6. Jouault, F. et al.: ATL: a QVT-like transformation language. In: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (2006). https://doi.org/10.1145/1176617.1176691
7. Burgueño, L., Cabot, J., Gerard, S.: The future of model transformation languages: an open community discussion. In: Journal of Object Technology 18.3. Ed. by Anthony Anjorin and Regina Hebig. The 12th International Conference on Model Transformations, 7:1-11. ISSN: 1660-1769 (2019). https://doi.org/10.5381/jot.2019.18.3.a7
8. Kehrer, T., Kelter, U., Ohrndorf, M. et al.: Understanding model evolution through semantically lifting model differences with SiLift. In: 28th IEEE International Conference on Software Maintenance (ICSM), pp. 638–641. IEEE (2012)
9. Kehrer, T., Taentzer, G. et al.: Automatically deriving the specification of model editing operations from meta-models. In: International Conference on Theory and Practice of Model Transformations, pp. 173–188. Springer (2016)
10. Rindt, M., Kehrer, T., Kelter, U.: Automatic generation of consistency-preserving edit operations for MDE tools. In: Demos@ MODELS 14 (2014)
11. Schultheiß, A., Bittner, P.M. et al.: 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. ACM, 27:1–27:6 (2020)
12. Schultheiß, A., Boll, A., Kehrer, T.: Comparison of graph-based model transformation rules. J. Object Technol. **19**(2), 1–21 (2020)
13. Hebig, R. et al.: 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 (2018). https://doi.org/10.1145/3236024.3236046
14. Rentschler, A. et al.: Designing information hiding modularity for model transformation languages. In: Proceedings of the 13th International Conference on Modularity. MODULARITY '14 (2014). https://doi.org/10.1145/2577080.2577094
15. Höppner, S., Tichy, M., Kehrer, T.: Contrasting Dedicated Model Transformation Languages vs. General Purpose Languages: A Historical Perspective on ATL vs. Java based on Complexity and Size: Supplementary Materials (2021). https://doi.org/10.18725/OPARU-38923
16. Götz, S., Tichy, M.: Investigating the origins of complexity and expressiveness in ATL transformations. In: The 16th European Conference on Modelling Foundations and Applications (ECMFA 2020) Journal of Object Technology 19.2. Ed. by Richard Paige and Antonio Vallecillo, 12:1-21 (2020). https://doi.org/10.5381/jot.2020.19.2.a12
17. Wieringa, R.J.: Design science methodology for information systems and software engineering. Undefined (2014). https://doi.org/10.1007/978-3-662-43839-8
18. Anjorin, A., Buchmann, T., Westfechtel, B., et al.: Benchmarking bidirectional transformations: theory, implementation, application, and assessment. Softw. Syst. Model. (SoSyM). (2019). https://doi.org/10.1007/s10270-019-00752-x
19. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **SE–2**(4), 308–320 (1976). https://doi.org/10.1109/TSE.1976.233837
20. Götz, S., Tichy, M., Kehrer, T.: Dedicated model transformation languages vs. general-purpose languages: a historical perspective on ATL vs. java. In: Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development—

Volume 1: MODELSWARD, INSTICC. SciTePress, pp. 122–135 (2021). https://doi.org/10.5220/0010340801220135
21. Steinberg, D., et al.: EMF: Eclipse Modeling Framework. Pearson Education (2008)
22. OMG.: Meta Object Facility (MOF) (2016). https://www.omg.org/spec/MOF
23. OMG.: Object Constraint Language (OCL) (2014). https://www.omg.org/spec/OCL/2.4/PDF
24. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–645 (2006)
25. Strüber, D. et al.: Henshin: a usability-focused framework for emf model transformation development. In: International Conference on Graph Transformation, pp. 196–208. Springer (2017)
26. Anjorin, A., Buchmann, T., Westfechtel, B.: The families to persons case. In: TTC'17 (2017)
27. Jouault, F.: ATL/Tutorials—Create a simple ATL transformation (2013). https://wiki.eclipse.org/ATL/Tutorials_-_Create_a_simple_ATL_transformation. Accessed 12 June 2021
28. SanchezCuadrado, J., et al.: Efficient execution of ATL model transformations using static analysis and parallelism. IEEE Trans. Softw. Eng. (2020). https://doi.org/10.1109/TSE.2020.3011388
29. Jabangwe, R., et al.: Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review. Empir. Softw. Eng. **20**(3), 640–693 (2015). https://doi.org/10.1007/s10664-013-9291-7
30. Weidmann, N. et al.: Incremental (unidirectional) model transformation with eMoflon::IBeX. In: Transformation, Graph (ed.) Esther Guerra and Fernando Orejas, pp. 131–140. Springer, Cham (2019) 978-3-030-23611-3
31. Cicchetti, A., et al.: JTL: a bidirectional and change propagating transformation language. In: Malloy, B., Staab, S., van den Brand, M. (eds.) Software Language Engineering, pp. 183–202. Springer, Berlin (2011)
32. Hinkel, G.: NMF: A Modeling Framework for the. NET Platform, KIT (2016)
33. Buchmann, T.: BXtend-a framework for (bidirectional) incremental model transformations. In: MODELSWARD, pp. 336–345 (2018)
34. Aniche, M.: Java code metrics calculator (CK) (2015). https://github.com/mauricioaniche/ck
35. Batory, D.S., Altoyan, N.: Aocl: a pure-java constraint and transformation language for MDE. In: MODELSWARD, pp. 319–327 (2020)
36. Singh, Y., Kaur, A., Malhotra, R.: Application of logistic regression and artificial neural network for predicting software quality models. In: Software Engineering Research and Practice, pp. 664–670 (2007)
37. Aggarwal, K.K., et al.: Investigating effect of design metrics on fault proneness in object-oriented systems. J. Object Technol. **6**(10), 127–141 (2007)
38. Pai, J.G., BechtaDugan, J.: Empirical analysis of software fault content and fault proneness using bayesian methods. IEEE Trans. Softw. Eng. **33**(10), 675–686 (2007). https://doi.org/10.1109/TSE.2007.70722
39. Guo, Y. et al.: An empirical validation of the benefits of adhering to the law of demeter. In: 2011 18th Working Conference on Reverse Engineering, pp. 239–243 (2011). https://doi.org/10.1109/WCRE.2011.36
40. GopalakrishnanNair, T.R., Selvarani, R.: Defect proneness estimation and feedback approach for software design quality improvement. Inf. Softw. Technol. **54**(3), 274–285 (2012). https://doi.org/10.1016/j.infsof.2011.10.001
41. Olbrich, S. et al.: The evolution and impact of code smells: a case study of two open source systems. In: 2009 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 390–400 (2009). https://doi.org/10.1109/ESEM.2009.5314231
42. Alshayeb, M., Li, W.: An empirical validation of object-oriented metrics in two different iterative software processes. IEEE Trans. Softw. Eng. **29**(11), 1043–1049 (2003). https://doi.org/10.1109/TSE.2003.1245305
43. Hinkel, G., Goldschmidt, T., et al.: Using internal domain-specific languages to inherit tool support and modularity for model transformations. Softw. Syst. Model. **18**(1), 129–155 (2019). https://doi.org/10.1007/s10270-017-0578-9
44. Kehrer, T., Kelter, U., Taentzer, G.: A rule-based approach to the semantic lifting of model differences in the context of model versioning. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp. 163–172. IEEE (2011)
45. Hinkel, G., Burger, E.: Change propagation and bidirectionality in internal transformation DSLs. Softw. Syst. Model. **18**(1), 249–278 (2019). https://doi.org/10.1007/s10270-017-0617-6
46. Boronat, A.: Expressive and efficient model transformation with an internal DSL of Xtend. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. MODELS '18. Copenhagen, Denmark: Association for Computing Machinery, pp. 78–88. ISBN: 9781450349499 (2018). https://doi.org/10.1145/3239372.3239386
47. Cuadrado, J.S., Molina, J.G., Tortosa, M.M.: RubyTL: a practical, extensible transformation language. In: Rensink, A., Warmer, J. (eds.) Model Driven Architecture-Foundations and Applications, pp. 158–172. Springer, Berlin (2006)
48. Akehurst, D.H. et al.: SiTra: simple transformations in java. In: Model Driven Engineering Languages and Systems, pp. 351–364. Springer (2006), ISBN: 978-3-540-45773-2
49. Horn, T.: Model querying with FunnyQT. In: Duddy, K., Kappel, G. (eds.) Theory and Practice of Model Transformations, pp. 56–57. Springer, Berlin (2013)
50. Zündorf, A. et al.: Story driven modeling libary (SDMLib): an Inline DSL for modeling and model transformations, the Petrinet-Statechart case. In: Sixth Transformation Tool Contest (TTC 2013), ser. EPTCS (2013)
51. Born, K., et al.: Analyzing conflicts and dependencies of rule-based transformations in henshin. In: Egyed, A., Schaefer, I. (eds.) Fundamental Approaches to Software Engineering, pp. 165–168. Springer, Berlin (2015)
52. Ehrig, H., Ermel, C., et al.: Semantical correctness and completeness of model transformations using graph and rule transformation. In: Ehrig, H. (ed.) Graph Transformations, pp. 194–210. Springer, Berlin (2008)
53. Leblebici, E. et al.: A comparison of incremental triple graph grammar tools. In: Electronic Communications of the EASST 67 (2014). https://doi.org/10.14279/tuj.eceasst.67.939
54. Bergmann, G., et al.: Viatra 3: a reactive model transformation platform. In: Kolovos, D., Wimmer, M. (eds.) Theory and Practice of Model Transformations, pp. 101–110. Springer, Cham (2015)
55. Martínez, S., Tisi, M., Douence, R.: Reactive model transformation with ATL. In: Science of Computer Programming 136, pp. 1–16 (2017). ISSN: 0167-6423. https://doi.org/10.1016/j.scico.2016.08.006. https://www.sciencedirect.com/science/article/pii/S016764231630106X
56. Le Calvar, T., et al.: Efficient ATL incremental transformations. J. Object Technol. **18**(3), 1–2 (2019)
57. van Amstel, M.F., van den Brand, M.G.J.: Using metrics for assessing the quality of ATL model transformations. In: MtATL@TOOLS (2011)
58. Vignaga, A.: Metrics for measuring ATL model transformations. In: MaTE, Department of Computer Science, Universidad de Chile, Tech. Rep (2009)
59. Getir, S. et al.: State elimination as model transformation problem. In: Transformation Tool Contest at the Conference on Software

Technologies: Applications and Foundations (TTC@STAF), pp. 65–73 (2017)

60. Beurer-Kellner, L., von Pilgrim, J., Kehrer, T.: Round-trip migration of object-oriented data model instances. In: Transformation Tool Contest at the Conference on Software Technologies: Applications and Foundations (TTC@STAF) (2020)
61. Batory, D.S., Altoyan, N.: Aocl: a pure-java constraint and transformation language for MDE. In: MODELSWARD, pp. 319–327 (2020)
62. Groner, R., et al.: A survey on the relevance of the performance of model transformations. J. Object Technol. **20**(2), 1–27 (2021). https://doi.org/10.5381/jot.2021.20.2.a5

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Stefan Höppner is a Ph.D. student at Ulm University. His research is focused on topics surrounding the development and evaluation of model transformation languages. In particular, he is interested in the advantages and disadvantages that these languages offer in contrast to general purpose languages. Prior to his work as a Ph.D. student he was a student of Software Engineering at Ulm University where he received his M.Sc. in.

Timo Kehrer is professor at Hum boldt-Universität zu Berlin (Germany), heading the Model-Driven Software Engineering Group at the Department of Computer Science. Before that, Kehrer was working as research assistant in the Software Engineering and Database Systems Group at University of Siegen (Germany) from 2011 to 2015, and as postdoctoral research fellow in the Dependable Evolvable Pervasive Software Engineering Group at Politecnico di Milano (Italy) from 2015 to 2016. He has active research interests in various fields of model-driven and model-based software and system engineering, with a particular focus on model evolution.

Matthias Tichy is full professor for software engineering at the University of Ulm and director of the institute of software engineering and programming languages. His main research focus is on model-driven software engineering, particularly for cyber-physical systems. He works on requirements engineering, dependability, and validation and verification complemented by empirical research techniques. He is a regular member of programme committees for conferences and workshops in the area of software engineering and model driven development. He is co-author of over 110 peer-reviewed publications.