Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Investigating the origins of complexity and expressiveness in ATL transformations

Stefan Götz[a] Matthias Tichy[a]

a. Institute of Software Engineering and Programming Languages, Ulm University, Germany

Abstract Model transformations provide an essential element to the model driven engineering approach. Over the years, many languages tailored to this special task, so-called model transformation languages, have been developed. A multitude of advantages have been proclaimed as reasons to why these dedicated languages are better suited to the task of transforming models than general purpose programming languages. However, little work has been done to confirm many of these claims. In this paper, we analyse ATL transformation scripts from various sources to investigate three common claims about the expressiveness of model transformation languages. The claims we are interested in assert that automatic trace handling and implicit rule ordering are huge advantages for model transformation languages and that model transformation languages are able to hide complex semantics behind simple syntax. We use complexity measures to analyse the distribution of complexity over transformation modules and to gain insights about what this means for the abstractions used by ATL. We found that a large portion of the complexity of transformations stem from simple attribute assignments. We also found indications for the usefulness of conditioning on types, implicit rule ordering and automatic trace resolution.

Keywords ATL; Complexity; Expressiveness; Model Transformation Languages; Analysis.

1 Introduction

Model transformations are a pivotal part of model-driven engineering (MDE) [SK03, Sch06]. This is also evident from the amount of transformation languages that have been proposed, i.e. ATL [JAB+06], Henshin [ABJ+10], ETL [KPP08], Viatra [BV06] and QVT [Kur07] just to name a few.

While the number of transformation languages and their features is ever increasing, little time is spent on empirical studies on the use of said languages [SCD17]. A fact

Stefan Götz, Matthias Tichy. *Investigating the origins of complexity and expressiveness in ATL transformations*. In Journal of Object Technology, vol. 19, no. 2, 2020, pages 12:1–21. doi:10.5381/jot.2020.19.2.a12

that is true not only for model transformation languages but any kind of DSL as evident from the results of [KBM16].

The authors of [SCD17], have shown that studying the use of transformation languages on code repositories such as the ATL Zoo [1] can provide insights into how a transformation language is used which can help developers with language evolution.

Such studies are also necessary because there is continual debate about whether dedicated model transformation languages are necessary at all [HSB+18, BCG19] since GPLs like Java can also be used for writing transformations and have been discussed as an alternative since the introduction of model transformations [SK03].

In the study described here, we apply these goals to the Atlas model Transformation Language (ATL) [JAB+06]. We are particularly interested in investigating transformation scripts to gather data concerning the following claims which have been made multiple times in literature:

H1: *Model transformation languages hide complex semantics behind simple syntax [JABK08, KCF14, SK03, GK03].*

H2: *Automatic handling and resolution of trace information by the transformation engines is a huge advantage of model transformation languages [JABK08, LR07, HGBR19].*

H3: *Model transformation languages allow for implicit rule ordering which can lessen the load on developers [JABK08, LR07].*

One thing that immediately stands out from the three claims is that they are intertwined. Automatic handling of traces and implicit rule ordering are both concepts that can hide certain semantics within the transformation engine. So to investigate their impact and provide insights into the complexity within model transformations as a whole we devised 5 research questions to focus our research on:

RQ1: *How is the complexity of ATL transformations distributed over multiple transformations and transformation components?* This question forms a basis data set for the following investigations. Its results can provide useful insights into where the complexity in ATL transformations originates from to provide starting points for more focused investigations. It can also help to uncover potential strengths and weaknesses of the abstractions used by ATL (*H1*).

RQ2: *When looking at the complexity distributions of individual transformation components, are there any salient characteristics?* ATL components such as the out-pattern consist of a set of bindings that assign values to the attributes of the output model. The question that arises from such structures is, whether the complexity of out-patterns stems largely from single complex bindings or a number of simpler bindings. With this research question we aim to investigate such effects which can indicate points where ATL does a good job of hiding complexity (*H1*)

RQ3: *How does the usage of refining mode impact the complexities of ATL modules?* ATLs refining mode was introduced to ease refinement transformations by allowing developers to only focus on the code generating modified elements while leaving all other elements unchanged. Accordingly, the complexity of

[1] `https://www.eclipse.org/atl/atlTransformations/`

re ning mode transformations should originate to large parts in re ning activities. Otherwise it would indicate that the re ning mode fails in supporting developers with model re nements. This in turn would be a counterexample to the claims made in H1.

RQ4: How large is the percentage of bindings that require trace-based binding resolution? Before being able to argue about the usefulness of trace information (H2) for model transformations it should be investigated to what extent their existence in uences a model transformation script. If only a small proportion of transformations utilize traces then maybe the development e ort for implicit trace handling is not worth it.

RQ5: What portion of ATL transformations use implicit rule ordering? The amount of implicitly ordered rules compared to manual rule ordering can be a good indication into whether the feature is well liked by developers hinting at an advantage over manual ordering.

To answer the research questions we selected a total of 33 ATL transformations from various sources to analyse. We use two sets of complexity measures based on [LKRSA18] to measure the complexity of ATL transformations. A meta-model representing the basic components of ATL modules is used to compile the complexity values together. Information about trace usage and rule ordering is taken directly from the models representing the ATL transformations.

The remainder of this paper is structured as follows: First in Section 2 an introduction into relevant aspects of ATL is given. Section 3 de nes the used complexity measures. Afterwards in Section 4 we present our extraction and analysis procedures. The results of our analysis are then presented in Section 5. Section 7 discusses potential threats to the validity of the described proceedings while Section 6 places the approach in the context of existing work. Lastly Section 8 concludes and proposes potential future work.

2 The Atlas Transformation Language (ATL)

Speci cations in ATL are organized in one of three kinds of so called*Units*. A unit is either a module, a library or a query. Depending on their type, units can consist of rules, helpers and attributes, which are a special kind of helper.

ATL uses the Object Constraint language (OCL) [OMG06] for both data types and expressions.

2.1 Modules

Modules are used to de ne transformations. ATL modules are made up of three segments (see Listing 1): the*module header*which de nes the modules name as well as the types of the input and output meta-models, a number of optional*imports* and a set of helper and rule de nitions.

```
1 module NAME
2   create OUT1:OUTTYPE1, ...
3   [from|refining] IN1:INTYPE1, ...
4
5   [uses LIBRARY]*
```

```
6   [RULEDEF|HELPERDEF]*
```

Listing 1 Structure of an ATL module

Libraries consist of a set of helper definitions. Libraries can be imported into modules.

Lastly, Queries are comprised of an import section, a query element and a set of helper definitions. Queries are used to define transformations from models to simple OCL types rather than output models.

2.2 Helpers and Attributes

Helpers allow developers to define outsourced expressions that can be called from within rules. Helper definitions can define a data type for which the helper is specified, called context. ATL also allows developers to define so called Attribute helpers. The main difference between attributes and helpers is that attributes do not accept parameters. Attributes serve as constants that are defined for a specific context.

The definition of both traditional helpers and attribute helpers follow the same syntax patterns (see Listing 2). The only difference lies in whether input parameters are defined.

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

Listing 2 Syntax to define Helpers

2.3 Rules

In ATL, rules are used to specify the transformation of input models into output models. There exist two main types of rules: called rules and matched rules. Matched rules enable a declarative way to define how a model element of a specific type is transformed into output model elements, while called Rules enable generation of target model elements from imperative code. Matched rules are executed automatically on all matching input model elements by the ATL engine.

Matched rules are comprised of four main sections (see Listing 3):

An In-Pattern which defines source model elements that are being transformed. In-Patterns can contain a filter expression which defines a condition that must be met for the rule to be applied.

An optional Using-Block that allows to define local variables.

The Out-Pattern which defines a number of output model elements that are created for the model element defined in the in-pattern when the rule is applied. Each output model element is defined by an Out-Pattern element which contains so called bindings that assign values to attributes of the model element.

And lastly an optional Action-Block which allows the specification of imperative code that is executed once the target elements have been created.

```
1 [lazy| unique lazy]? rule NAME {
2   from
3     INVAR : INTYPE [(CONDITION)]*
4   [using {
5     [VAR : VARTYPE = EXPR;]+
```

```
  }]?
  to
    [OUTVAR : OUTTYPE {
      [ATR <- EXPR,]+
    },]+
  [do {
    [STATEMENT;]*
  }]?
}
```

Listing 3 Syntax to de ne matched rules

Apart from regular matched rules there are alsolazy rules. They are de ned by adding the key word lazy in front of a matched rule de nition. Lazy rules are executed only when explicitly called for a speci c model element that matches the rules type and lter expression. Lazy rules can be called multiple times on the same model element to produce multiple distinct output elements.

Unique lazy rules , de ned through the unique lazy key words, change this behaviour. Instead of producing a new model element for each call, unique lazy rules always return the same output element when called on the same input model element.

Lastly, called rules are de ned in a similar fashion to matched rules (see Listing 4). The main di erence between the two being that called rules do not contain an In-Pattern and allow the de nition of required parameters.

```
rule NAME([PARAMETER,]*) {
  [using {
    [VAR : VARTYPE = EXPR;]+
  }]?
  to
    [OUTVAR : OUTTYPE {
      [ATR <- EXPR,]+
    },]+
  [do {
    [STATEMENT;]*
  }]?
}
```

Listing 4 Syntax to de ne called rules

2.4 Re ning mode

The re ning mode is a special execution mode for ATL rules which is intended to assist developers with refactoring models, i.e., endogenous transformations.

Normally, the ATL engine only produces output model elements for input elements on which rules are executed on. When using the re ning mode however, the ATL engine executes all rules on matching input elements and produces a copy of all unmatched elements. This way developers are able to focus solely on the re ning part of their refactoring e orts according to the language developers.

3 Complexity Measures

There exist several approaches for measuring complexity of model transformation languages and ATL in particular ([DRDRIP15, Vig09, TSMGD+11, KGBH10, LKRSA18]). Most of these approaches use a simple metric that relates the number of transformation components such as rules or helpers to the complexity of a transformation module. In our opinion, however, the number of rules or helpers alone does not capture the complexity of model transformations well enough. For that reason, we opted to adopt the complexity measure proposed by [LKRSA18] which includes not only the number of transformation components but also the complexity of expressions used within the transformation.

In the following, the complexity measures will be explained.

3.1 Syntactic complexity

The syntactic complexity c( ) of a transformation speci cation is de ned based on the complexity of expressions and activities within the de ned transformation [LKRSA18]. The general idea behind it being that the complexity of each construct is comprised of a static value for the construct itself plus the sum of the complexities of its contained elements.

The complexity of a module as de ned in Listing 1 would be comprised of the sum of the complexities for its contained helper de nitions and rule de nitions. The complexity of rules, de ned as shown in Listings 3 and 4, is then comprised of the complexity of their contained from-block (In-Pattern), the to-block (Out-Pattern), the using-block and do-block plus a static value of 1 for the rule itself.

The complexity of In-Patterns is de ned by their contained lter expression and a static value for the construct itself, while the complexity of Out-Patterns is de ned by a static value for the construct as well as the sum of the complexities of all contained Out-Pattern elements and their contained bindings. An overview over the most important complexity measure de nitions can be found in Table 1 for expressions and Table 2 for activities/structural elements[2].

We adopted the complexity measure with slight modi cations since we disagreed with certain de ned values. The following adjustments were made to the de nition from [LKRSA18]:

First, the complexity of helpers was adapted to also include the complexity of their context. The reason for this change being the fact that the context of a helper has to be considered when trying to understand its function. Furthermore, in our opinion there is no di erence between attribute and operation helpers, the additional, static complexity attributed to both types of helper de nitions was aligned at 1. For this the static complexity of attribute helpers was reduced from 3 to 1 and that of operation helpers was increased from 0 to 1.

Action blocks were given an additional static complexity value of 1 which was missing from the de nitions of [LKRSA18]. This aligns it with the static complexity that is attributed to all elements contained within rules, i.e. In-Patterns, Out-Patterns and Using-blocks.

The complexity attributed to operation calls was increased to 1 to align it with that of attribute and navigation calls. In our opinion calling an operation on an object is

[2] Full de nitions can be found in https://spgit.informatik.uni-ulm.de/stefan.goetz/mtl-complexities/blob/master/ATL/transformations/qvt/transforms/complexity.qvto

just as complex as accessing one of its attributes. For the same reason the complexity for collection operation calls was also increased to 1 as well.

Table 1 De nitions of expression complexity measure based on [LKRSA18].

Expression e	Complexity c(e)
Numeric, boolean or String value	0
Identi er iden	1
Attribute call source:attr	c(source) + 2
Operation call source:op(p1; ::)	$2 + c(source) + \sum_i c(p_i)$
Operator call e1 op e2	c(e1) + c(e2)
CollectionOperation call source > op(p1; ::)	$2 + c(source) + \sum_i c(p_i)$
if e 1 then e2 else e3 endif	c(e1) + c(e2) + c(e3) + 1
let v : t = e1 in e2	c(t) + c(e1) + c(e2) + 4
CollectionExpression Col f e1; ::g	$1 + \sum_i c(e_i)$
Primitive Type (Integer,String,...)	1
Collection Type Col(t)	1 + c(t)

3.2 Computational complexity

The computational complexity is an extension of the syntactic complexity. Its goal is to more closely capture the underlying complexity of transformation de nition with respect to outsourced expressions and called transformation rules. To achieve this, the complexity of Operation Call s is calculated by taking the complexity of the called operation into account instead of adding a static value regardless of the called operation. For example given a helpersample of syntactic complexity 12, the call sample() has a syntactic complexity of 2 whereas its computational complexity amounts to 12.

Moreover the complexity of used variables is also resolved by taking the de nition expression of the variable into account instead of using a static value of 1.

4 Methodology

Apart from the selection of the ATL transformation modules to analyse, we strongly oriented our proceedings along the research questions from section 1.

4.1 Module Selection

The selection of ATL modules was aimed to achieve a wide spread of transformations based on their source, purpose and size in terms of lines of code. We also aimed to achieve an even distribution of modules that use the re ning mode and modules that do not.

For this purpose, we searched GitHub for ATL projects by using the search string `ATL transformation` and included all novel (meaning not present in the ATL zoo) transformations for which we were also able to nd the input and output meta-models

Table 2 De nitions of complexity measure for ATL elements/activities based on [LKRSA18]. ATL elements are capitalized while expression elements are written in lower case.

ATL element A	Complexity $c(A)$
Module $H_1;::;R_1;::$	$\sum_i c(H_i) + \sum_i c(R_i)$
Helper helper context c def : n : t = e	c(c) + c(t) + c(e)
MatchedRule rule N f From Using To Dog	c(From) + c(To) + c(Do) + c(Using)
CalledRule rule N (p) f Using To Dog	c(To) + c(Do) + c(Using)
VariableDe nition n : t = e	c(t) + c(e) + 3
InPattern from s : t (f)	c(f) + c(t) + 3
OutPattern o : t f $B_1;::$g	$c(t) + \sum_i c(B_i) + 2$
Binding n <- e	c(e) + 2
ActionBlock do f Sg	c(S)
S1;S2	c(S1) + c(S2)
if e then S1 elseS2	c(e) + c(S1) + c(S2) + 1
for v : e do S	c(e) + c(S) + 1
Binding Statement v <- e	c(v) + c(e) + 1

Table 3 Meta-data about the analysed transformation modules.

Data	minimum	average	maximum	total
LOC	39	408	1364	13455
Rules	1	14	55	460
Helpers	0	11	74	376
Bindings	2	112	487	3695

since those were required for parts of our analysis(see Section 4.4). This resulted in a total of 16 transformation modules. Additionally we included the R2ML2XML transformation from [vAvdB11] and the families2persons transformation from the ATL zoo because it is a widely used example for model transformations. We then supplemented the set of transformations with transformations from the ATL zoo to try and achieve an even distribution between modules that use the re ning mode and modules that do not.

The result was a set of 33 ATL transformations (some meta-data about the transformations can be found in Table 3). Of those 33 transformations, 15 use the re ning mode of ATL while 18 are exogenous transformations. A complete overview over the selected transformations, including names and sources can be found under https://spgit.informatik.uni-ulm.de/stefan.goetz/mtl-complexities/blob /master/ATL/resources/input/cases/justifications .

4.2 RQ1,2: How is the complexity of ATL transformations distributed over multiple transformations and transformation components and are there any salient characteristics?

To be able to collect and analyse complexity data of ATL transformations and relevant elements thereof a meta-model was constructed[3]. Its structure was designed to be able to break down the full representation of an ATL transformation into the basic components that make up ATL transformations as described in Section 2. With this structure it is also possible to investigate where the complexity of entire ATL modules and rules originate from, e.g. whether a rule is complex because of its size or due to a few complex contents like lter expressions. The design of the meta-model followed the principles of abstraction and pragmatics. Compared to the ATL meta-model our developed meta-model focuses solely on those parts of the ATL transformations we are interested in and provides an easy way to track their complexity and the origin thereof.

To transform transformation modules into a model of the presented meta-model and to calculate the complexities of its components along the way, a QVT-o transformation[4] was de ned. Its correctness was evaluated using unit tests: A test module containing at least one of each activities/expressions for which a complexity value can be calculated was de ned. The complexity values for each element was calculated manually based on the previously introduced complexity de nition. Afterwards the results of the transformation were manually compared with the manually calculated complexity values. Discrepancies between the complexity values were investigated and corrected.

In order to collect data for analysis, the tested transformation was applied to the 33 ATL transformations.

Apart from the raw complexity data, we resorted to using several diagrams such as histograms, violin and alluvial plots as well as code snippets to investigate the complexity distribution, both syntactical and computational, of ATL transformations.

In order to better understand the meaning behind the complexity values example code snippets for each component were extracted from the 33 selected transformation modules. The code snippets were selected so that their complexity values correspond to the components median complexity within the data set. One such code snippet can be seen in Listing 5. All used snippets can be found under https://spgit.informatik.uni-ulm.de/stefan.goetz/mtl-complexities/tree/master/ATL/resources/input/medians .

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

Listing 5 Helper with a syntactic complexity corresponding to the median of all helper complexities.

[3] the meta-model can be found under https://spgit.informatik.uni-ulm.de/stefan.goetz/mtl-complexities/-/tree/master/ATL/metamodels/ATLComplexity/model

[4] https://spgit.informatik.uni-ulm.de/stefan.goetz/mtl-complexities/blob/master/ATL/transformations/qvt/transforms/complexity.qvto

4.3 RQ3: How does the usage of re ning mode impact the complexities of ATL modules?

As explained in Section 1, we also intended to analysed ATL modules using the re ning mode as a example of how transformation languages hide semantics.

To do so, we used the 15 selected transformation modules that use re ning mode and analysed their complexities separately and in comparison to those modules not using the re ning mode.

4.4 RQ4: How large is the percentage of bindings that require trace-based binding resolution?

To investigate the usefulness of trace-based binding resolution (and thus to an extent that of implicit trace management) we resorted to analysing how often it is used in transformation modules. A high proportion of trace-based resolutions used would then indicate their usefulness. Since trace-based binding resolution only happens along reference types of the input and output elements we extracted all reference types per module element for all output meta-models. For this we used a simple Java-program that given an Ecore- le would produce a list of reference types for each contained EClass.

Afterwards the bindings within all selected transformation modules were analysed for usage of the extracted reference types. The amount of bindings that use traces compared to simple assignments was then analysed on the basis of these results.

4.5 RQ5: What portion of ATL transformations use implicit rule ordering?

Similar to the trace usage, the usefulness of implicit rule ordering can be indicated by the distribution of implicitly ordered transformation elements compared to explicitly ordered ones.

Called and lazy matched rules all get explicitly ordered by developers when calling them while matched rules enable the ATL transformation engine to traverse the source model and implicitly order their execution. Thus the ratio of matched rules to called and lazy rules gives an indication into how relevant implicit rule ordering is for model transformations.

Data for this analysis can be gathered from both the complexity distributions from RQ 1 as well as directly from the number of de nitions.

5 Result Summary and Analysis

We present the results of our analysis in this section in accordance with the research questions posed in Section 1.

5.1 RQ1: How is the complexity of ATL transformations distributed over multiple transformations and transformation components?

Figures 1 and 2 show alluvial plots over the distribution of syntactic complexity and computational complexity respectively of module elements within ATL transformation modules. They display how much of the complexity of all investigated transformations originate in which components beginning with the modules themselves following the

de nitions down to the contained expressions and the static value associated with each component.

Interestingly, while making up nearly 45% of all top level de nitions, Helpers only contribute to roughly 18% of the total complexity of a transformation module [5]. The largest portion of complexity is attributed to matched rules which contribute to over 3/4 of the total complexity of transformation rules while accounting for 53% of all top level de nitions. And lastly called Rules , which are not widely represented in our data sets, while making up about 1% contribute to 5% of the overall complexity of modules.

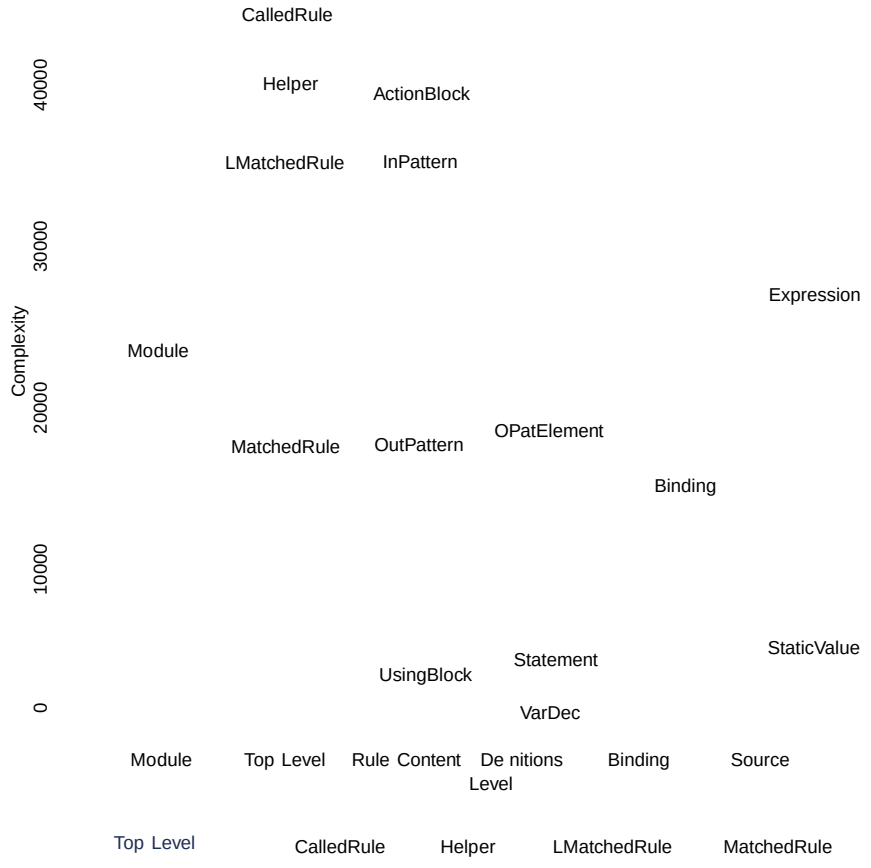

Figure 1 Distribution of syntactic complexity over all ATL modules.

Another observation that can be made from Figure 1, is that about 80% of the syntactic complexity of (lazy) Matched-rules stems from their Out-Patterns while only 15% come from In-Patterns and a nearly negligible 5% originate in action- and using blocks . Following this trend downwards 73% of the complexity of these rules stems from their contained bindings, i.e assigning a value to

[5] the raw data can be found under https://spgit.informatik.uni-ulm.de/stefan.goetz/mtl-complexities/tree/master/ATL/data

attributes of the output model element. Meaning most e ort in transformations is spent not in selecting the correct model elements to transform but simply assigning the output values (see Section 5.2 for a more detailed discussion). This e ect is still present when looking at the computational complexity distribution (as shown in Figure 2) which rules out the possibility that the e ect is created by outsourcing of lter conditions in In-Patterns through helpers. This leads us to:

Observation 1: Over half of the e ort spent in writing ATL transformations is spent assigning values to the output model.

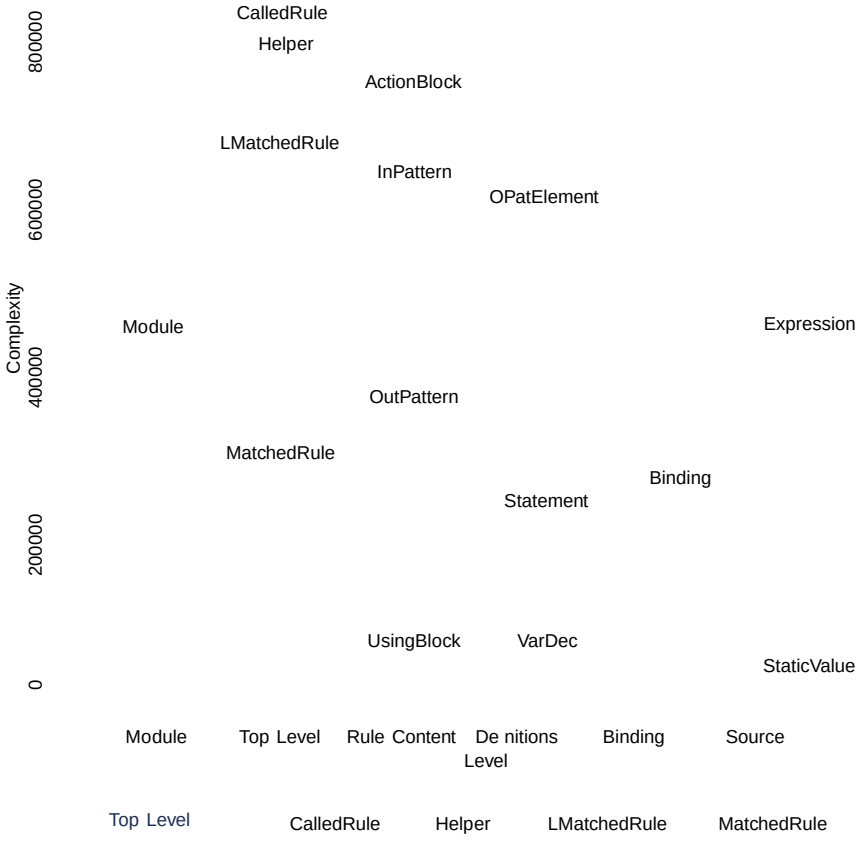

Figure 2 Distribution of computational complexity over all ATL modules.

Furthermore the 15% of matched rules syntactic complexity that comes from In-Patterns shows that conditioning the application of transformation is a relevant task for model transformations. The low proportion especially when considering the computational complexity rather suggests that the conditioning on types (as opposed to lter conditions without pre conditioning on types) which ATL does for all matched rules by default alone already provides a useful abstraction for model transformations. This assumption is supported by the fact that about 25% of all matched rules get by with only using the standard type conditioning without any

additional lter expression in the In-Pattern . Of those 25% only 12% (which therefore constitute only 3% of all matched rules) are trivial transformation rules. Simple transformations in the context of this paper mean transformations that contain no lter condition and only assign attributes from the input model element to the output model element without doing any additional operations. The fact that a large portion of transformations get by with only the default conditioning on types in ATL leads us to:

Observation 2: Conditioning on types provides an abstraction well suited for model transformations.

5.2 RQ2: When looking at the complexity distributions of individual transformation components, are there any salient characteristics?

Figure 3 Syntactic complexity distribution of Bindings.

As previously mentioned, the proportion of the complexity of bindings within transformations also stands out in Figure 1. Bindings alone make up over half of the complexity of transformations, a trend, that persists even when looking at the computational complexity of transformation modules. Interestingly, the complexity within bindings is very unevenly distributed. Figure 3, which shows a violin and box plot for the syntactic complexity distribution of bindings (note the logarithmic scale of the y axis), illustrates this. The majority of all bindings have a syntactic complexity 5 (b_5 = 60%). This corresponds to directly accessing an attribute of an object as shown in Listing 6, calling a helper on said object or accessing a global attribute (thisModule.attribute).

Further analysis shows that a total of 93% of all bindings with a syntactic complexity of 5 do indeed stem from direct accesses of attributes of the input model element (b_{5a} = 93%). Only 2% are global attribute accesses and the last 5% originate from helper calls on the source model element. This is also indicated by the

fact that a majority of bindings have a computational complexity of 7 which can only correspond to accessing attributes on input elements. In summary, this leads us to:

> Observation 3: Over half ($b_5 - b_{5a} = 56\%$) of all bindings are used to map one attribute of an input model element to one attribute of an output model element.

Adding to this point, only 5% of bindings with a syntactic complexity of 5 stem from trivial transformations, i.e., transformations that simply map attributes from an input element to an output element without doing any meaningful filtering or modification of the content. This reinforces Observation 3 since we can rule out that the majority of bindings with complexity of 5 stem from trivial transformations which do by definition only contain bindings of this or lower complexity .

Looking at ATL modules as a whole Observation 3 means that 33% of their total syntactic complexity comes from the activity of copying input model attributes to output model attributes.

```
1 rule MedianBinding {
2   from s : Families!Member
3   to t : Persons!Female (
4     fullName <- s.firstName
5   )
6 }
```

Listing 6 Rule containing a binding with a syntactic (computational) complexity of 5 (7)

That much of the complexity of transformation modules comes from bindings means that the main effort when writing model transformations in ATL consists of defining how the output should look which is actually one of the main goals of model transformation languages. This in turn suggests that ATL does a good job in abstracting away other tasks in model transformation such as model traversal, conditioning on types as shown in Section 5.1, tracing and rule ordering to which we will come in Sections 5.4 and 5.5.

5.3 RQ3: How does the usage of refining mode impact the complexities of ATL modules?

Given the observations from the previous sections we would expect that the syntactic complexity distribution of bindings to deviate away from 5 (and the computational complexity form 7) since the refining mode is designed to enable developers in focusing only on the refining part of the transformation.

In the transformations investigated for this paper this is however not the case as can be seen in Figure 4, the median syntactic complexity of bindings remains 5 and that of computational complexity remains 7.

This indicates the usefulness of the changes made to the refining mode with the introduction of the 2010 ATL compiler. Since 2010 refining mode allows real in-place transformations which means that rules only need to specify changes to elements while all the other elements remain untouched. And because the main effort in the investigated transformations, which were all defined for ATL compilers prior to 2010, is spent on copying attributes from the input model element (98% of all bindings) to its output counterpart, newer versions of the ATL compiler would heavily reduce this necessary overhead, allowing developers to focus solely on actually refining models. To us this suggests that the current versions of ATLs refining mode can significantly reduce unnecessary overhead for refining transformations. There is also an observation

to be made from this:

Observation 4: GitHub and especially the ATL Zoo lack samples of ATL transformations using the refining mode with compiler versions at least as current as 2010.

Figure 4 Complexity distribution of Bindings in refining mode.

Furthermore, InJJJ Patterns in refining mode are, on average about twice as complex as in non refining modules. Moreover only a small portion (7%) of In-Patterns do not contain a filter expression at all compared to 1/3 of In-Patterns in non refining mode.

This leads us to two additional observations:

Observation 5: When refining models, filters are more heavily used than when transforming between different meta-models.

Observation 6: Filter expressions are more complex in refining mode due to having to select elements with more specific properties.

5.4 RQ4: How large is the percentage of bindings that require trace-based binding resolution?

About 15% of all bindings in the analysed transformation modules require traces. While this makes it apparent that traces are less frequently required than one would expect, it still demonstrates their necessity since 15% is not a negligible proportion. This leads us to:

Observation 7: Bindings that require traces constitute a significant part of the model transformations considered.

It is also worth mentioning that while such trace resolution can save developers substantial amounts of time they can also be a source of errors.

Considering the complexity of the bindings that require traces also reveals something interesting. About half of all bindings that require traces have a syntactic complexity

of 5 and computational complexity of 7. This shows how well automatic trace handling can hide complexity. The developer can simply access the input model element that is supposed to be transformed into the correct output model element and the transformation engine handles resolving and referencing. Would the developer have to take care of this process manually both syntactic and computational complexity would be signi cantly higher since this would require identifying and accessing the corresponding output model element through additional code.

5.5 RQ5: What portion of ATL transformations use implicit rule ordering?

In the ATL modules analysed for this study, a total of 460 rules are de ned. Of those 364 or 79% are matched rules, 84 or 18% are lazy matched rules which need to be invoked to transform model elements and only 12 or 2% are called rules.

Our results, deviate slightly from the results found by [SCD17] but still reveal the same preference trend of ATL developers:

Observation 8: Developers strongly prefer matched rules over lazy matched rules and called rules.

Since matched rules allow for implicit model traversal and rule ordering this can indicate that these concepts provide good support for transformation developers. This is also evident from the fact that the proportion of Out Pattern complexity (both syntactic and computational) to Action Block complexity is far more balanced in called rules than in (lazy) matched rules (see Figures 1 and 2) again indicating that called rules require more structural code such as calling other rules and conditioning.

6 Related Work

In [DRDRIP15], the authors analyse the impact of input and output meta-models on, amongst other things, the complexity of ATL transformations. For this purpose they use a number of meta-model metrics and correlate these with metrics for ATL transformations using Spearman's rank correlation coe cient. Their ndings include a high correlation between the number of structural features of the output meta-models and the number of used bindings in an ATL transformation module. An insight which can be re ected upon in our results. In contrast to the complexity measures applied in this work however, the measures proposed for complexity in their work is con ned to the number of structural features such as bindings or helpers of ATL transformations rather than the complexity of their structure and contained expressions. Which is not to say that the applied measure is not indicative of the complexity of transformations rather that it is only part of what makes a transformation complex in our opinion.

The authors of [vAvdB11] propose the usage of cyclomatic complexity to measure the complexity of helpers. They also envision incorporating the complexity of the contained OCL expression into its complexity measure. Similarly to [DRDRIP15], they also use the number of di erent transformation components as metrics for measuring ATL transformations. The described metrics are applied to seven transformations. And the resulting values are then related to quality attributes, based on the assessment of nineteen experts, such as understandability, maintainability and conciseness using Kendall's $_b$ correlations. Notable results include a signi cant correlation between the number of transformation rules and conciseness and the number of out-patterns and understandability. In comparison to this, we try to draw direct conclusions about the

structure and structure of transformations from our gathered data instead of about quality features.

In [Vig09] the complexity of ATL transformations is related to a variety of introduced metrics. Most of the related metrics are once again quanti cations of di erent components within ATL modules such as thenumber of matched rulesor average number of lters used in rules. They also relate thecyclomatic complexity to complexity much like [vAvdB11]. As previously mentioned we believe that the number of components are only part of what makes transformations complex which is why the used complexity measure in this paper also incorporates the complexity of expressions.

The numbers of ATL transformation components have also been used in [SMGD+11] to make comparisons between several transformation modules to investigate the feasibility of applying transformations to transformation modules. The authors concede that the applied metrics need further research and development and predict that such measures could assist with identifying aspects of ATL transformations to optimize.

Similarly [KSW+13] analysed the ATL Zoo with the goal to gain insights about the frequency of use of reuse mechanisms. For this the authors devised a semi-automated process to extract and analyse projects from the ATL zoo. They found that reuse mechanisms are exclusively used within a transformation and that helpers are the most frequently used reuse mechanism while only little rule inheritance is used. In contrast to their work, our focus does not directly relate to reuse mechanisms although the computational complexity was introduced in part to account for the outsourcing of complexity due to reuse mechanisms.

7 Threats to validity

This section addresses the potential threats to validity identi ed for the performed study.

The transformations evaluated for the purpose of this study were chosen from various sources to reduce the in uence of programming habits of individual transformation engineers. Consequently the purposes and characteristics of the transformations vary immensely. To be able to compare transformation modules using re ning mode with modules that do not use re ning mode we also aimed to use a similar amount of respective transformation modules. While this strengthens the external validity of our comparison it can potentially lead to a reduction in the external validity of our other ndings since an even distribution of re ning and non re ning modules is potentially less representative of the overall ATL ecosystem. Given the selection of transformation modules it is also not possible to draw representative conclusions about model transformation languages in general but rather for ATL speci cally.

There is of course a discussion to be held about the complexity measure used. As discussed in Section 6 most research uses the number of elements as basis for complexity measures. We and [KRSA18] argue that this alone does not fully cover the complexity of transformations. The syntactic complexity measure used in this study uses the complexity of expressions and activities as de ned in Tables 2 and 1. The number of elements are also taken into account in these de nitions but do not constitute the majority of the complexity value of an ATL transformation this is reserved to the complexity of expressions used within the transformation modules as evident from Figure 1. While we are missing a formal validation of the measures used we believe that this indicates their overall usefulness. The computational complexity is then a natural extension of the syntactic complexity to more closely resemble the

actual complexity that is hidden in operation calls in expressions.

8 Conclusion and Future Work

In this work we presented our results of analysing ATL modules to provide insights into three common claims about the advantages of model transformation languages, namely that transformation languages hide complex semantics behind simple syntax, that automatic trace handling in transformation languages is advantageous and that implicit rule ordering supports developers in de ning transformations.

For this purpose we used two complexity measures to investigate how complexity is distributed over ATL transformation modules which we applied to a total of 33 modules. We also analysed the proportions of matched rules compared to other types of rules and the proportion of bindings that require trace information to be resolved.

We found, that while transformations can get complex, the complexity originates mainly in de nitions of how the output models should be populated rather than how the transformation should be executed. To us this provides an indication for how well ATL abstracts away from certain tasks necessary for model transformation such as model traversal, rule ordering and trace handling.

We have also shown that conditioning on types is well suited for model transformations since a total of 22% of all non-trivial matched rules get by with only ltering on types. This also provides a clear example why implicit rule ordering can be bene cial for model transformation de nitions since developers can simply de ne to which kind of input model element a transformation should apply and the transformation engine handles execution.

This is further supported by the fact that we found that nearly 80% of all de ned rules are matched rules which make use of exactly this mechanism.

Next we analysed required trace information in bindings. We came to the conclusion that while bindings that do require trace information are outweighed by those that do not, they still constitute a signi cant portion of model transformations. And while this suggests that automatic trace handling is advantageous further research is necessary to more precisely capture its impact.

Lastly we compared the complexities of transformation modules using the re ning mode with those that do not. We found that while the complexity of matched rules de ned in a re ning module is much higher, the increase in complexity can be attributed to an increase in simple bindings. A fact we were able to attribute to the use of older ATL compilers which did not allow in-place re nements.

For future work, we are interested in repeating the described proceedings on transformations written in general purpose programming languages. While the resulting values can not be compared directly, the complexity distributions can be used to gain insights into where the complexity in these transformation de nitions lie. Which we believe can produce further contributions to the discussion of GPLs vs MTLs for de ning model transformations.

References

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place emf model transformations. In Model Driven Engineering

Languages and Systems, Berlin, Heidelberg, 2010. doi:https://doi.org/10.1007/978-3-642-16145-2_9 .

[BCG19] Loli Burgueño, Jordi Cabot, and Sébastien Gérard. The future of model transformation languages: An open community discussion. Journal of Object Technology, July 2019. doi:10.5381/jot.2019.18.3.a7 .

[BV06] András Balogh and Dániel Varró. Advanced model transformation language constructs in the viatra2 framework. In Proceedings of the 2006 ACM Symposium on Applied Computing, New York, NY, USA, 2006. doi:10.1145/1141277.1141575 .

[DRDRIP15] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Mining correlations of atl model transformation and metamodel metrics. In 2015 IEEE/ACM 7th International Workshop on Modeling in Software Engineering 2015. doi:10.1109/MiSE.2015.17.

[GK03] J. Gray and G. Karsai. An examination of dsls for concisely representing model traversals and transformations. In 36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the, Jan 2003. doi:10.1109/HICSS.2003.1174892 .

[HGBR19] Georg Hinkel, Thomas Goldschmidt, Erik Burger, and Ralf Reussner. Using internal domain-speci c languages to inherit tool support and modularity for model transformations. Software & Systems Modeling Feb 2019. doi:https://doi.org/10.1007/s10270-017-0578-9 .

[HSB+18] Regina Hebig, Christoph Seidl, Thorsten Berger, John Kook Pedersen, and Andrzej W¡sowski. 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. doi:10.1145/3236024.3236046 .

[JAB+06] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. Atl: A qvt-like transformation language. In Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications, New York, NY, USA, 2006. doi:10.1145/1176617.1176691 .

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. Science of Computer Programming, 2008. doi:https://doi.org/10.1016/j.scico.2007.08.002 .

[KBM16] Tomaº Kosar, Sudev Bohra, and Marjan Mernik. Domain-speci c languages: A systematic mapping study. Information and Software Technology, 2016. doi:https://doi.org/10.1016/j.infsof.2015.11.001.

[KCF14] Filip Krikava, Philippe Collet, and Robert France. Manipulating Models Using Internal Domain-Speci c Languages. In Symposium On Applied Computing, Gyeongju, South Korea, March 2014. doi:10.1145/2554850.2555127.

[KGBH10] Lucia Kapová, Thomas Goldschmidt, Steffen Becker, and Jörg Henss. Evaluating maintainability with code metrics for model-to-model transformations. In *International Conference on the Quality of Software Architectures*, 2010. doi:https://doi.org/10.1007/978-3-642-13821-8_12.

[KPP08] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The epsilon transformation language. In *Theory and Practice of Model Transformations*, Berlin, Heidelberg, 2008. doi:https://doi.org/10.1007/978-3-540-69927-9_4.

[KSW+13] Angelika Kusel, Johannes Schönböck, Manuel Wimmer, Werner Retschitzegger, Wieland Schwinger, and Gerti Kappel. Reality check for model transformation reuse: The atl transformation zoo case study. In *Amt@ models*, pages 1–11, 2013.

[Kur07] Ivan Kurtev. State of the art of qvt: A model transformation language standard. In *International Symposium on Applications of Graph Transformations with Industrial Relevance*, 2007. doi:https://doi.org/10.1007/978-3-540-89020-1_26.

[LKRSA18] Kevin Lano, Shekoufeh Kolahdouz-Rahimi, Mohammadreza Sharbaf, and Hessa Alfraihi. Technical debt in model transformation specifications. In *Theory and Practice of Model Transformation*, Cham, 2018. Publishing. doi:10.1007/978-3-319-93317-7_6.

[LR07] Michael Lawley and Kerry Raymond. Implementing a practical declarative logic-based model transformation engine. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, New York, NY, USA, 2007. doi:10.1145/1244002.1244216.

[OMG06] OMG. Object constraint language (ocl), April 2006.

[SCD17] Gehan MK Selim, James R Cordy, and Juergen Dingel. How is atl really used? language feature use in the atl zoo. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2017. doi:10.1109/MODELS.2017.20.

[Sch06] Douglas Schmidt. Guest editor's introduction: Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY*, 03 2006. doi:10.1109/MC.2006.58.

[SK03] S. Sendall and W. Kozaczynski. Model transformation: the heart and soul of model-driven software development. *IEEE Software*, Sep. 2003. doi:10.1109/MS.2003.1231150.

[TSMGD+11] José Barranquero Tolosa, Oscar Sanjuán-Martínez, Vicente García-Díaz, B Cristina Pelayo G-Bustelo, and Juan Manuel Cueva Lovelle. Towards the systematic measurement of atl transformation models. *Software: Practice and Experience*, 2011. doi:https://doi.org/10.1002/spe.1033.

[vAvdB11] Marcel F van Amstel and MGJ van den Brand. Using metrics for assessing the quality of atl model transformations. In *MtATL@ TOOLS*, 2011.