How is ATL Really Used? Language Feature Use in the ATL Zoo

Gehan M. K. Selim
McMaster University, Hamilton, ON, Canada
Cairo University, Cairo, Egypt
Email: selimg@mcmaster.ca

James R. Cordy
Queen's University
Kingston, ON, Canada
Email: cordy@cs.queensu.ca

Juergen Dingel
Queen's University
Kingston, ON, Canada
Email: dingel@cs.queensu.ca

***Abstract*—Studies of code repositories have long been used to understand the use of programming languages and to provide insight into how they should evolve. Such studies can highlight features that are rarely used and can safely be removed to simplify the language. Conversely, combinations of features that are frequently used together can be identified and possibly replaced with new features to improve the user experience. Unfortunately, this kind of research has not been as popular in Model Driven Development (MDD). More specifically, using repositories of model transformations (in any language) to understand how the features of these languages are used has not been investigated much, despite its potential benefits. In this paper, we study the use of the ATL model transformation language in an ATL transformation repository. We identify three research questions aimed at providing insight into how ATL's features are actually used. Using the TXL source transformation language, we implement a parser-based analyzer to extract information from the ATL Zoo. We use this information to answer these research questions and provide additional observations based on manual inspection of ATL artifacts.**

***Index Terms*—Model transformations, MDD, ATL, TXL**

1. Introduction

To deal with the increasing complexity of software and its development, Model Driven Development (MDD) has been proposed as an alternative to code-centric software development. MDD is a software development methodology where *models* (i.e., software abstractions) constitute the main artifact of the software development process, and *model transformations* are the programs that are used to map between different models and from models to code.

While research interest in MDD has increased, some MDD aspects are less commonly investigated despite their potential benefits. For example, thousands of empirical studies on the use of programming languages have been conducted, and they are commonly used to provide insight into code-centric software development. By comparison, while certainly some empirical analysis has been done on the use of MDD, e.g., [1], [2], only a few studies [3], [4] have targeted model transformations. In part this is due to the limited availability of large model and model transformation repositories, but not exclusively so. By contrast, many large code repositories are publicly available, allowing extensive empirical studies on programming languages and software systems, e.g., [5], [6], [7], [8]. Our aim is to highlight the importance of such studies for MDD in general and model transformation in particular, by studying the use of transformation language features. Such studies help transformation language developers to evolve languages by adding new features, dropping obsolete features, and improving other language artifacts (e.g., documentation and grammar).

To demonstrate the usefulness of such studies for transformations, we investigate the use of features of the Atlas model Transformation Language (ATL) [9], [10], a textual model-to-model transformation language with imperative and declarative constructs. We study an ATL model transformation repository, the ATL Zoo [11], to understand how language features are used. To do so, we build a parser-based analyzer using the TXL source transformation language [12], extract various features from the transformations in the ATL Zoo, and discuss our findings. We choose ATL as the subject of our study due to its wide use, its well established community, and the public availability of an ATL transformation repository (i.e., the ATL Zoo).

Our contribution in this study is a step towards understanding how ATL is actually used. We do so by structuring our investigation around three main research questions:

RQ1: *Which ATL features are used in the Zoo?* This question will help ATL developers identify how to evolve ATL by keeping features that are heavily used and deprecating ones that are rarely used.

RQ2: *In which context are the ATL features used?* Certain ATL features are optional and can be used with different constructs. Understanding where such features are used will highlight whether some optional language features are necessary or if they can be deprecated to simplify the language.

RQ3: *How often is the imperative subset of ATL used?* Many of the available model transformation verification tools operate only on the declarative subset of ATL, e.g., [13], [14], [15]. Identifying the frequency of use of imperative ATL constructs will help confirm whether there is a pressing need to refactor popular imperative *patterns* to declarative equivalents to enable verification using these tools.

This paper is organized as follows: Section 2 overviews ATL's features; Section 3 presents our data set (i.e., the ATL Zoo) and the language used to analyze it (i.e., TXL); Section 4 answers the three research questions by extracting

```
1 module modName;
2     create outM1:outMM1, ..., outMn:outMMn
3         [from|refining] inM1:inMM1, ..., inMm:inMMm ;
4   [uses libraryRef]*
5   [rulesAndHelpers]*
```

Listing 1. Syntax of a Module in ATL

features from the Zoo using TXL; Section 5 discusses the threats to validity; Section 6 discusses additional observations; Section 7 surveys related work; and Section 8 concludes and presents future work.

2. The ATL Model Transformation Language

ATL specifications consist of *units*, which in turn contain *rules*, to transform and create output model elements, and *helpers*, which play a role similar to functions in programming languages. ATL has both *declarative* and *imperative* constructs, which can be mixed and matched depending on the complexity of the transformation. The declarative paradigm is preferable due to its simplicity and conciseness. Imperative constructs support transformations that are too complicated to implement in a purely declarative style. More information on ATL can be found in [9], [10].

2.1. ATL Units

ATL has three kinds of units: *modules*, *libraries*, and *queries*, each defined in a separate file. Modules can contain both rules and helpers, whereas libraries and queries can contain only helpers.

Modules: A module corresponds to an ATL transformation, and we use the two terms interchangeably in this study. Thus, an ATL module generates output model elements (conforming to some target metamodel) from input model elements (conforming to some source metamodel).

A module has four sections: a header, an import section, a set of helpers (if any), and (one or more) rules (Listing 1). Line 1 defines the module's name (*modName*). Lines 2-3 define the output models to create (*outMi*), the target metamodels to which the output models conform (*outMMi*), the input models (*inMj*), and the source metamodels to which the input models conform (*inMMj*). Line 3 also defines the execution *mode* as either normal (denoted by keyword **from**) or refining (denoted by **refining**). Normal mode is intended for exogenous transformations (those with different source and target metamodels) [16]. In this mode output models only contain elements that are explicitly created by transformation rules. Refining mode, by contrast, is intended for endogenous transformations (those using the same source and target metamodel) [16]. In this mode any input model elements not matched by transformation rules are automatically copied to the output model. Normal mode can also be used to build endogenous transformations, but any input model elements to be preserved must be explicitly copied. We refer to modules that use the **from** and **refining** keywords as *regular* and *refining* modules respectively.

The *import* section (line 4) of the module header specifies the libraries to be imported and used in the module. Any number of libraries can be imported using **uses** statements.

```
1 rule ruleName {
2   from inElem : inType [in modelin]?  [(condition)]?
3  [using { localVariables } ]?
4   to
5     outElem1 : outType1 [in modelout1]? (bindings1),
6        ...
7     outElemN : outTypeN [in modeloutN]? (bindingsN)
8  [do { imperativeStatements } ]? }
```

Listing 2. Syntax of Matched Rules in ATL

A set of one or more *rules* (Section 2.2) and zero or more *helpers* (Section 2.3) follows the import section.

Libraries: A library defines a set of helpers. A library cannot be executed independently, but can be imported into other units (modules, queries, and other libraries) that can reuse the helpers in the library.

Queries: A query has an *import* section, a mandatory *query* element, and optional helpers. A query element specifies a transformation from one (or more) input model(s) to any OCL type, often a primitive data type such as a string or boolean value. In the ATL Zoo [11], queries are mainly used to implement transformations from input models to code, XML, or other textual output documents.

2.2. ATL Rules

There are four kinds of rules in ATL: *matched rules*, *lazy rules*, *unique lazy rules*, and *called rules*.

Matched Rules: These specify how an input model element is transformed to one or more output model elements. Matched rules have four sections: a *source pattern*, a *target pattern*, a *local variables section*, and an *imperative block*. Only the source and target patterns are mandatory; the local variables and imperative block are optional [17]. The syntax of matched rules is shown in Listing 2.

After declaring the rule's name as *ruleName* (line 1), the **from** keyword specifies a *source pattern* (line 2), which consists of the type of element (*inType*) to match in the input model (*modelin*), the variable to which the matched element is bound (*inElem*), and a boolean *condition* or filter on the matched element. The **using** keyword specifies the local variables section (line 3), where variables that are local to the rule are defined and initialized.

The **to** keyword specifies the *target pattern* (lines 4-7) which consists of *simple* and/or *iterative* target pattern elements. Target pattern elements are output elements to create when the source pattern is matched in the input. In Listing 2, the target pattern contains *N* simple target pattern elements (*outElem1* ... *outElemN*). Each simple target pattern element has the type of element (*outTypeI*) to create in the output model (*modeloutI*), the variable to which the created element is bound (*outElemI*), and bindings (*bindingsI*) to initialize the output elements from input elements and local variables. While simple target pattern elements allow creating one output element at a time, iterative target pattern elements allow creating several elements of the same type at a time. While still supported by ATL, iterative target pattern elements have been deprecated since they break traceability links [17].

The **do** keyword specifies the imperative block (line 8) and can be used to specify any additional computation to be

```
1 rule ruleName (parameters) {
2  [using { localVariables } ]?
3  [to
4     outElem1 : outType1 [in modelout1]? (bindings1),
5       ...
6     outElemN : outTypeN [in modeloutN]? (bindingsN) ]?
7  [do { imperativeStatements } ]? }
```

Listing 3. Syntax of Called Rules in ATL

performed on the local variables or the target pattern using ATL's imperative constructs (Section 2.4).

Matched rules in a module are automatically executed without being called. For practical reasons they are normally executed in order, although this is not guaranteed.

For each matched rule, the input model is traversed and for every element in the input that matches the source pattern, a corresponding target pattern is produced in the output (i.e., one cannot run a matched rule on selective matching elements). Hence, a matched rule is executed only once for any matching element in the input.

Lazy Rules: Unlike matched rules, lazy rules are rules that are executed only when invoked for a specific matching element in the input model (as opposed to being automatically executed for all matches), and can be invoked multiple times for that element. Their syntax is similar to matched rules, but using the keywords **lazy rule**.

Unique Lazy Rules: Like lazy rules, these are executed only when invoked for a specific matching element, and can be invoked multiple times for that element. Unlike lazy rules, which recompute the target element each time they are invoked, unique lazy rules always return the same target element for each matching element. Unique lazy rules are declared using the keywords **unique lazy rule**.

Called Rules: These are used to create output elements without matching a source pattern. They are the only rules that can take parameters and can only be invoked in an imperative block. Called rules are executed only when invoked, unlike matched rules which are automatically executed for any matching input element. The syntax of called rules is shown in Listing 3. Called rules do not have a source pattern; they only have an optional local variable section (line 2), an optional target pattern (lines 3-6), and an optional imperative block (line 7). In other words, a called rule can have only a target pattern or only an imperative block, with the option of using local variables. The local variables section, target patterns, and imperative blocks in called rules have the same syntax and semantics as their equivalents in matched rules.

Two special cases of called rules are *entrypoint* and *endpoint* called rules. While a module can have many called rules, it can have a maximum of one entrypoint and one endpoint called rules. As opposed to called rules, entrypoint and endpoint called rules are implicitly invoked at the beginning and end of the transformation execution, respectively. Entrypoint and endpoint called rules are declared using the keywords **entrypoint** and **endpoint**. To differentiate between the three called rule types, we will refer to them as *regular*, *entrypoint*, and *endpoint* called rules.

2.2.1. Rule Inheritance. ATL supports *inheritance* between matched, lazy, and unique lazy rules, denoted by the keyword **extends**, as in **rule** A **extends** B { ... }. Rules inherit from super-rules (e.g., B in the example), which can be either *abstract* (denoted by the keyword **abstract**), which means a non-executable placeholder whose sole purpose is to be inherited from, or simply another executable rule. The set of rules that inherit from one another form a hierarchy. Rule inheritance works by first matching the super-rule, and then matching the sub-rules on the input that was matched by the super-rule. The deepest match in the inheritance hierarchy is the rule applied. While original ATL allows for only one **extends** clause per rule, EMFTVM [18] adds support for multiple inheritance.

```
1 helper [context cType]? def : hName (params) : returnType
      = declarativeExp;
```

Listing 4. Syntax of Functional Helpers in ATL

2.3. ATL Helpers

ATL has functional and attribute helpers. Functional helpers are declarative methods which take parameters, have return values, and can be invoked at any point in the module. The syntax of functional helpers is shown in Listing 4.

A functional helper has a name (e.g., *hName*), parameters (e.g., *params*), a return type (e.g., *returnType*), a context (i.e., the type of element from which this helper can be called, e.g., *cType*), and its declarative body (e.g., *declarativeExp*). Since functional helpers take parameters, their value can change in the same context based on the parameters passed. Hence, functional helpers are reevaluated each time they are called.

Attribute helpers are declarative attributes or functions that do not take parameters, i.e., their values are constant in a specific context. They are normally evaluated only the first time they are invoked, but this is not guaranteed. The syntax of attribute helpers is similar to that of functional helpers, except that they do not take parameters. The main use of attribute helpers in the ATL Zoo [11] is similar to the use of global variables in programming languages. Attribute helpers can be used to compute a global constant once to be available to all transformation rules, or it can be used as a global variable that can be updated (reassigned to) by rules.

2.4. Declarative Vs. Imperative ATL

In ATL, functional and attribute helpers are completely declarative. Based on the ATL documentation [17], matched, lazy, and unique lazy rules are considered declarative and can optionally have an imperative block (denoted by the keyword **do**). All called rule types (regular, entrypoint, and endpoint) are considered imperative.

In this study, we only consider a rule to be imperative if it has an imperative block, and can thus have global side effects. Thus, we consider a regular called rule with a target pattern and a local variables section to be declarative, since it does not have an imperative block. Whereas a matched rule with an imperative block is considered imperative.

ATL has declarative expressions that can be used in the declarative body of helpers and rules. Declarative ATL expressions include **if** expressions, **let** expressions, constants,

Transformation scenarios	99
ATL files	288
ATL rules	3,213
Total LOC	82,100

TABLE 1. OVERVIEW OF THE CLEANED ATL ZOO DATA SET

Scenario size	ATL files	ATL rules	LOC
Largest	20	203	4,909
Smallest	1	2	24
Average	3	32	829

TABLE 2. SIZE OF THE CLEANED ATL ZOO DATA SET

functional and attribute helper call expressions, operation calls on ATL data types, and **iterate** expressions on collection data types [17]. ATL also offers three imperative statements that can be used in imperative blocks of all rules: assignment statements, conditional **if** statements, and **for** statements. Details on declarative and imperative ATL constructs can be found in the ATL documentation [17].

3. Methodology

We discuss the data set used in this study (Section 3.1), and the language used to analyze it (Section 3.2).

3.1. Data Set

We used the ATL Zoo [11] to analyze the usage of ATL's features. The ATL Zoo is an ATL repository containing 103 transformation *scenarios*. A scenario can contain one or more modules (i.e., transformations, Section 2.1), their source and target metamodels, sample input and output models, and optional queries and libraries. Thus, some scenarios contain the files pertaining to one transformation, while others contain the files of transformation chains or versions of the same transformation. We refer to the transformation scenarios by their names as they appear in the ATL Zoo [11].

Data Cleaning. After manually inspecting the scenarios in the Zoo, we identified some necessary data cleaning steps.

Removing Duplicates: The *CatalogueModelTransformations* scenario consists only of duplicates of 14 other existing scenarios. Thus, this scenario was deleted from our data set. Further, the *UML2MOF* and *MOF2UML* scenarios both consist of the same two transformations: a transformation from MOF to UML, and its inverse. Thus, in the MOF2UML scenario we only kept the transformation from MOF to UML, and the inverse transformation was deleted. Similarly, in the UML2MOF scenario we only kept the transformation from UML to MOF.

Syntax Errors: The *KM32CONFATL* scenario has syntax errors that prevent it from being parsed by the system we use for analysis (i.e., TXL, Section 3.2). In particular, it uses the ATL keyword **uses** as a variable name. We renamed the variable to 'uses_' to resolve the error.

Non-existent Files: Two scenarios in the Zoo, *OpenBlueLab2UML* and *SimplePDL2Tina*, are empty. Further, the *QVT2ATLVM* scenario has no ATL files; it consists only of an Ant script that runs a compiled transformation file. Since our analysis processes ATL source units only, these scenarios were removed from our data set.

```
1  define module
2      'module [identifier] ; [NL]
3        'create [list oclModel+] [refining_or_from]
4            [list oclModel+] ;
5        [libraryRef*]
6        [moduleElement*]
7  end define
8  define refining_or_from
9      'refining | 'from
10 end define
11 define moduleElement
12     [attributeHelper] | [functionalHelper] | [rule_]
13 end define
```

Listing 5. A snippet of the TXL grammar for ATL [20] showing the syntax of ATL modules

Overview of the Cleaned Data Set: Tables 1 and 2 summarize the final, cleaned data set. As Table 1 shows, the data set has 99 scenarios, comprising 288 '.atl' files with 3,213 rules and 82,100 pretty-printed lines of code (LOC). Table 2 shows the size range of the scenarios, where size is measured in number of files, number of rules, and LOC. As Table 2 shows, the scenarios in the cleaned ATL Zoo vary from one to 20 files, with an average of three files. The scenarios range in size from two rules (comprising 24 LOC) to 203 rules (4,909 LOC), with an average of 32 rules (829 LOC) per scenario. We use this cleaned ATL Zoo (available at [19]) to analyze the usage of ATL's features in Section 4.

3.2. Feature Extraction Using TXL

Our analysis is based on a precise ATL parser implemented in the TXL source transformation language [12]. The parser is based on an ATL grammar adapted for TXL [20] from the Eclipse ATL syntax [21]. Listing 5 shows a snippet of the ATL grammar adapted for TXL, demonstrating the syntax of modules. We verified that the grammar and the ATL Zoo are consistent by ensuring that all 288 files in the 99 scenarios of the Zoo parse correctly using the grammar.

Our ATL analyzer uses the parser to perform feature extraction on two levels. At the first level, we extract high level ATL features, such as the frequency of using different constructs (e.g., different rule and helper types), deprecated constructs (e.g., iterative target patterns), assignments, and inheritance. At the second level, we use *cascaded* feature extraction to extract more detailed information from the high level ATL features extracted at the first level. Listing 6 shows a snippet of the multi-level feature extraction performed by our analyzer. Lines 2-3 show how we perform feature extraction at the first level by, for example, extracting all regular called rules (denoted by *RegCallRules*) from the ATL program (denoted by *P*). The TXL extract function [^] extracts a list of all of the instances of the target nonterminal type (in this case [*calledRule*]) in the parameter parse (in this case the ATL program *P*). Lines 4-5 count the number of extracted regular called rules (*NumRegCallRules*), using the TXL [*length*] function to count the number of items in the extracted list. The remainder of Listing 6 shows how we perform feature extraction at the second level. For example, lines 7-8 extract the local variables sections (denoted by

```
% Extract & Count Regular Called Rules
construct RegCallRules [calledRule*]
  _ [^ P]
construct NumRegCallRules [number]
  _ [length RegCallRules] [putp "NumRegCallRules"]
% Extract & Count Using Clauses in Called Rules
construct UsingRegCallRules [using_clause*]
  _ [^ RegCallRules ]
construct NumUsingRegCallRules [number]
  _[length UsingRegCallRules][putp"NumUsingRegCallRules"]
% Extract & Count Target Patterns in Called Rules
construct ToRegCallRules [outPattern*]
  _ [^ RegCallRules ]
construct NumToRegCallRules [number]
  _ [length ToRegCallRules][putp "NumToRegCallRules"]
% Extract & Count Imperative Blocks in Called Rules
construct DoRegCallRules [actionBlock*]
  _ [^ RegCallRules ]
construct NumDoRegCallRules [number]
  _ [length DoRegCallRules][putp "NumDoRegCallRules"]
```

Listing 6. A snippet of our TXL analyzer using the ATL grammar

UsingRegCallRules) from the list of regular called rules extracted at the first level. Lines 9-10 then count the number of extracted local variables sections (**using** clauses) in regular called rules (*NumUsingRegCallRules*). Similarly, lines 12-15 extract and count the number of target patterns, and lines 17-20 the number of imperative blocks, in regular called rules. Due to space limitations, we do not show more complicated TXL functions used to extract more detailed features such as the number of functional helpers using attribute helpers that are reassigned.

4. Analysis

We explore the research questions introduced in Section 1 by executing our TXL-based ATL analyzer on the 99 scenarios in our cleaned data set. Fig. 1 summarizes our analysis of the use of different ATL features at two levels. At the first level, Fig. 1 shows the percentage of modules, libraries, and queries that contain at least one instance of each feature on the y-axis. Since all features in Fig. 1 (except for library imports, and functional and attribute helpers) can only occur in modules, Fig. 1 shows the percentage of the 225 modules in the Zoo that have at least one instance of these features. Library imports and functional and attribute helpers can occur in all ATL units, according to ATL's grammar [21]. Thus, Fig. 1 shows the percentage of modules, libraries, and queries that have at least one instance of each of these three features. At the second level, Fig. 1 shows the percentage of the 99 scenarios that have at least one instance of each of the features.

RQ1. Which ATL features are used in the ATL Zoo?

ATL Units: Fig. 2 shows the classification of the 288 files based on their ATL unit type. Only 5.6% of the files (16 files) are queries, 16% (47 files) are libraries, and 78% (225 files) are modules. Regular modules using the **from** keyword comprise 70% (202) of the files, while 8.0% (23 files) are refining modules that use the **refining** keyword. Thus, the majority of the ATL units are modules (i.e., transformations), with regular modules being the most widely used in the Zoo. Queries are the least used ATL units.

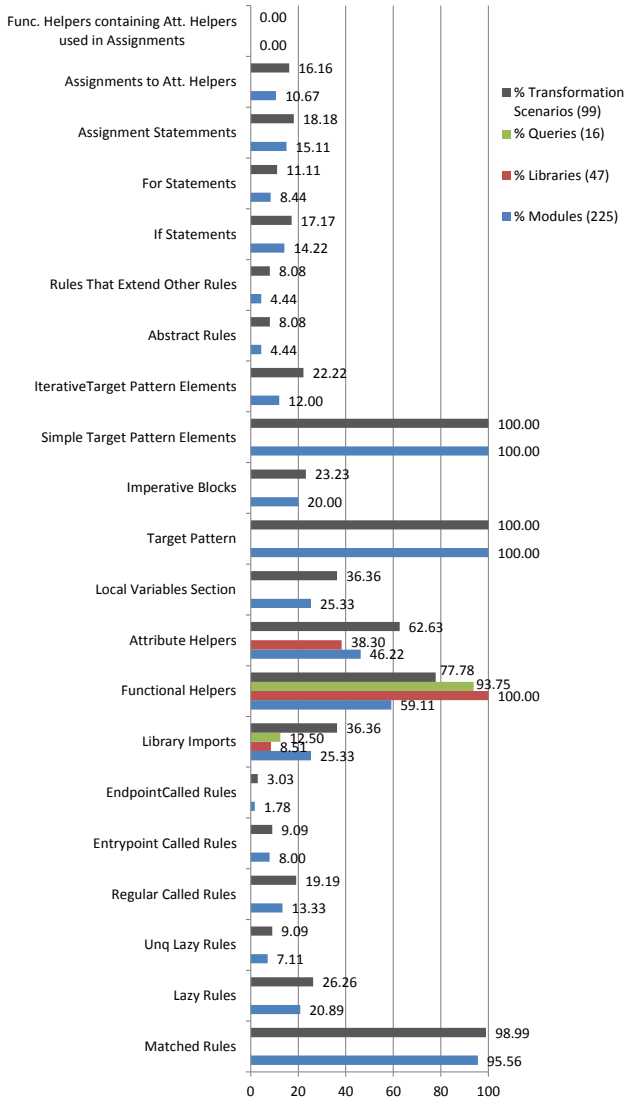

Figure 1. Percentage of modules, libraries, queries, and scenarios (x-axis) that have atleast one instance of different ATL features (y-axis)

We further categorize the 225 modules as *endogenous, exogenous*, or *both*. Endogenous modules are transformations between models that conform to the same metamodel, i.e., the source and target metamodels are identical [16]. Endogenous transformations can be either refining modules (i.e., modules that use the **refining** keyword) or regular modules (i.e., modules that use the **from** keyword) that manipulate the same source and target metamodels. Exogenous modules are regular modules or transformations between models that conform to different source and target metamodels [16]. In this study, we identify a third type of modules; those that are simultaneously endogenous and exogenous. The ATL Zoo has endogenous, refining modules that have one or more output models conforming to the source metamodel, as well as one or more output models that conform to a different metamodel. Based on our correspondence with AtlanMod's team, it is clear that the ATL semantics allows for endogenous, refining modules

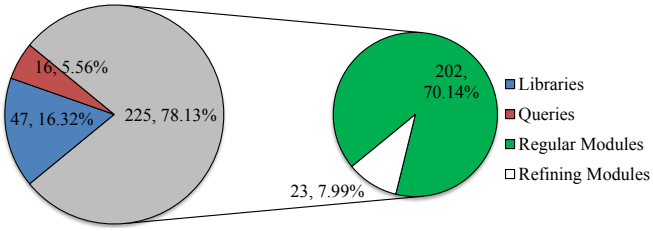

Figure 2. Classification of the 288 ATL Zoo files by ATL unit type

with additional *side-effect* output models that conform to a metamodel other than the source metamodel. As shown in Fig. 3, 82% of the modules (184 files) are exogenous and 17% of the modules (39 files) are endogenous. Thus, most transformations in the Zoo are exogenous, i.e., regular modules that use different source and target metamodels. Only two modules (0.89%) are both endogenous and exogenous, and exist in one scenario (*ATL2Tracer*).

Summary: *The majority (70%) of the units in the Zoo are regular modules and a minority (5.6%) are queries. The majority of modules (82%) are exogenous.*

ATL Rules: The 225 modules contain a total of 3,213 rules of all types. As shown in Fig. 4, most (87%) of the rules in the Zoo are matched rules, with other rule types used much less frequently. Lazy rules are the second most widely used (4.8%), followed by called rules (4.6%), and unique lazy rules (3.7%). Regular called rules are the most widely used called rules (comprising 3.9% of all rules), followed by entrypoint (0.56%), and endpoint called rules (0.12%).

Since modules are the only units that can contain rules, we investigate how widespread the use of each rule type is by extracting them from the 225 modules, while ignoring libraries and queries. As shown in Fig. 1, the majority (96%) of the 225 modules have at least one matched rule, making them the most widespread rule type in the Zoo. Other rule types are used much less commonly. Lazy rules are the second most widely used rule type, with 21% of the 225 modules having at least one lazy rule. Regular called rules follow, where 13% of the 225 modules have at least one regular called rule. Less than 10% of the modules have at least one entrypoint called rule, unique lazy rule, or endpoint called rule. Endpoint called rules are the least used rules, appearing in only 1.8% of modules.

Fig. 1 also shows the percentage of the 99 scenarios that have at least one instance of each rule type, showing a similar trend to modules. Most scenarios (99%) have at least one matched rule. Lazy rules are the second most widely used in scenarios (26%), followed by regular called rules (19%), entrypoint called rules (9.1%), unique lazy rules (9.1%), and endpoint called rules (3.0%).

Summary: *The majority (87%) of rules in the Zoo are matched rules and the minority (0.12%) are endpoint called rules. Matched rules are also the most widespread rule type in both modules and scenarios, with over 95% of each having at least one matched rule. Endpoint called rules are the least widespread, with less than 4%.*

Helpers and Library Imports: Since functional helpers, attribute helpers, and library imports can occur in all ATL

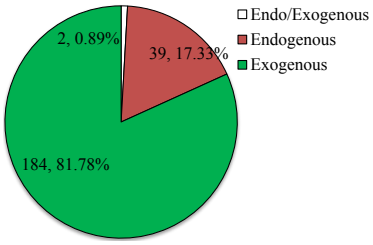

Figure 3. Classification of the 225 ATL Zoo modules as endogenous, exogenous, or both

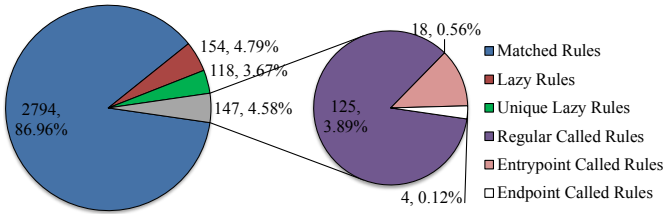

Figure 4. Classification of the 3,213 rules in the Zoo based on their type

units, we extract these three features from modules, libraries, and queries. The 288 ATL units (i.e., modules, libraries, and queries) in the Zoo contain a total of 1,876 functional helpers, 595 attribute helpers, and 86 library imports.

As Fig. 1 shows, a large percentage of modules (59%), libraries (100%), and queries (94%) contain at least one functional helper. A smaller percentage of modules (46%) and libraries (38%) contain at least one attribute helper, whereas none of the queries contain an attribute helper. An even smaller percentage of modules (25%), libraries (8.5%), and queries (13%) import at least one library. Fig. 1 shows a similar trend at the level of transformation scenarios, where 78% of the scenarios have at least one functional helper, 63% of the scenarios have at least one attribute helper, and only 36% of the scenarios import at least one library. The high usage of functional helpers indicates that developers often adopt good development practices, such as modularizing their transformations, to improve maintainability and promote code reuse.

Summary: *Functional helpers are more widespread across all units and scenarios than attribute helpers, whereas library imports are rarely used in the ATL Zoo.*

RQ2. In which context are the ATL features used?

Many ATL features can be used with any rule type. These features include optional rule sections (i.e., local variables section, target pattern, imperative block), kinds of target pattern elements (i.e., simple and iterative), and inheritance features (i.e., *abstract rules* and rules that *extend* other rules). We use Fig. 5 to identify the percentage of rules of each rule type that use these features.

Optional Rule Sections: Fig. 5 shows the percentage of each rule type that has a local variables section, a target pattern, and an imperative block. We do not count the percentage of rules that have a source pattern, since source patterns are mandatory in three rule types (matched, lazy, and unique lazy rules), and are not present in three rule types (regular, entrypoint, and endpoint called rules).

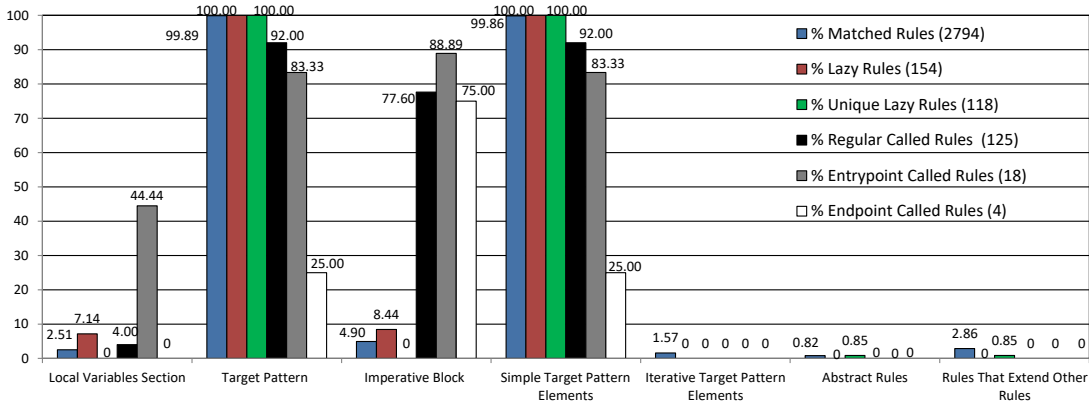

Figure 5. Percentage of each rule type (y-axis) that uses different ATL features (x-axis). Each rule type in the legend is followed by the number of rules of that type in the Zoo.

As shown in Fig 5, 44% of entrypoint called rules (8 of 18) have a local variables section. The five other rule types rarely use a local variables section. Only 7.1% of lazy rules (11 of 154), 4.0% of regular called rules (5 of 125), and 2.5% of matched rules (70 of 2,794) have a local variables section. None of the unique lazy rules or endpoint called rules in the Zoo have a local variables section.

With respect to target patterns, almost all lazy rules (100%), unique lazy rules (100%), and matched rules (99.9%) in the Zoo have target patterns (Fig. 5). A large percentage of the regular (92%) and entrypoint called rules (83%) have a target pattern, while only 25% of endpoint called rules have a target pattern.

Imperative blocks are rarely used in matched (4.9%) and lazy rules (8.4%), and are not used at all in unique lazy rules. Most of the regular (78%), entrypoint (89%), and endpoint called rules (75%) use imperative blocks. Given that imperative blocks are mainly found in the called rule types (i.e., regular/entrypoint/endpoint called rules), this finding is consistent with the ATL documentation [17] which considers all called rule types to be imperative, while matched, lazy, and unique lazy rules are considered declarative.

Fig. 1 shows that all (100%) 225 modules and 99 scenarios in the Zoo have at least one target pattern. This is not surprising since, as previously stated, target patterns are mandatory in three rule types according to ATL's documentation [17]. Local variables and imperative blocks are used in a smaller percentage of modules and scenarios. Specifically, 25% of the modules and 36% of the scenarios have at least one local variables section. Similarly, 20% of the modules and 23% of the scenarios have at least one imperative block.

Summary: *While local variables are used in 44% of entrypoint called rules, they are rarely used in other rule types. For rules in which target patterns are optional, most actually have target patterns, except endpoint called rules, where only 25% have target patterns. Imperative blocks are used in over 75% of all called rules, but are rarely used in matched, lazy, and unique lazy rules.*

Target Pattern Elements: Target patterns in rules can have any number of simple or iterative target pattern elements. Iterative target pattern elements, which are allowed only in matched rules, have been deprecated. Fig. 5 shows the percentage of each rule type that has at least one simple or one iterative target pattern element in their target patterns. Almost all matched (99.9%), lazy (100%), and unique lazy rules (100%) have at least one simple target pattern element. The majority of regular called rules (92%) and entrypoint called rules (83%) have at least one simple target pattern element. Only 25% of endpoint called rules have at least one simple target pattern.

Although iterative target patterns have been deprecated, we found 121 of these elements. Specifically, 1.6% of matched rules (44 of 2,794) have at least one iterative target pattern element.

With respect to how widespread the use of target pattern elements is, Fig. 1 shows that all 225 modules and 99 scenarios have at least one simple target pattern element, while only 12% of the modules and 22% of the scenarios have at least one iterative target pattern element. The use of the deprecated iterative target pattern elements can be avoided by explicitly not supporting them in ATL.

Summary: *Simple target pattern elements are highly used in all rule types, except for endpoint called rules (25%). While iterative target patterns have been deprecated, they are still used in 1.6% of matched rules in the Zoo.*

Inheritance Features: According to the ATL grammar [20], [21], regular, entrypoint, and endpoint called rules do not support inheritance (i.e., use the **abstract** and **extends** keywords). Thus, Fig. 5 shows that 0% of these rules abstract or extend other rules. For the remaining rule types, Fig. 5 shows that less than 3% of the matched and unique lazy rules in the Zoo use inheritance, while none of the lazy rules use inheritance.

Fig. 1 shows that inheritance is rarely used in the Zoo, with only 4.4% of the 225 modules and 8.1% of the 99 scenarios containing at least one *abstract* or *extended* rule. This could be for many reasons, e.g., transformations in the Zoo may not require the use of inheritance or, it could be that object-oriented concepts are still not widely adopted in the model transformation community.

Summary: *Inheritance is rarely used in the Zoo; less than 3% of matched and unique lazy rules use inheritance.*

RQ3. How often is the imperative subset of ATL used?

As mentioned in Section 1, this question is of interest since many verification tools operate only on the declarative subsets of transformation languages [13], [14], [15], [22], [23]. This is mainly due to the complexity of verifying imperative languages in general. Thus, answering this question can enable us to identify how pressing the need is to develop verification tools that can manipulate imperative ATL code, or to devise methods to automatically refactor imperative ATL to its declarative equivalents to facilitate verification.

In this paper, we consider a rule to be imperative only if it has an imperative block, where an imperative block can have three imperative constructs: **if**, **for**, and assignment statements (Section 2.4). As shown in Fig. 1, 20% of the 225 modules (45 modules) and 23% of the 99 scenarios (23 scenarios) have at least one imperative block.

With respect to imperative constructs, Fig. 1 shows that assignments are the most widely used, where 15% of the modules and 18% of the scenarios have at least one assignment. **if** statements follow, where 14% of the modules and 17% of the scenarios have at least one **if** statement. **for** statements are the least used in the Zoo, where 8.4% of modules and 11% of scenarios have at least one **for**.

We also identified how often functional helpers use attribute helpers that are reassigned to in assignment statements. In Section 2.3 we saw that the main use of attribute helpers in the Zoo is similar to the use of global variables, where they can be reassigned to by any rule. While functional helpers are declarative (Section 2.4), they can no longer be considered as such if their code uses reassigned attribute helpers, because they could return a different output for the same input, depending on the current value of the attribute helper. Since nothing in ATL prevents functional helpers from using attribute helpers that are reassigned to, we are interested in whether developers use this practice. As Fig. 1 shows, while 11% of modules and 16% of scenarios have at least one assignment to an attribute helper, none of the modules (and hence, scenarios) contain functional helpers that use attribute helpers that are reassigned.

Summary: *Less than 24% of the modules and scenarios are considered imperative, i.e., have at least one imperative block. Assignments are the most widely used imperative constructs, followed by* ***if*** *and* ***for*** *statements. Further, 11% of the modules and 16% of the scenarios use attribute helpers as global variables that are reassigned to by transformation rules. However, none of the functional helpers in the modules (and hence, scenarios) use such attribute helpers, which reflects the desire of ATL developers to keep their functional helpers truly declarative.*

5. Threats To Validity

We identify four threats to the validity of our study. First, due to its broad range of applications and public availability, we conducted our study on the ATL Zoo. While the transformations in the Zoo vary widely in size and purpose (Tables 1 and 2), many of these transformations are notably old which might affect why, for example, deprecated iterative patterns where still used in these transformations. The Zoo is also not intended to be a repository of ATL practice, rather a demonstrative catalogue of example applications. Thus, we cannot say if the transformations in the Zoo are truly representative of current, typical use of ATL in industry or academia. This is a general problem for empirical work and can be solved either by the creation of an open source repository similar to those available for many programming languages, or by obtaining a more representative set of ATL transformations, for instance, by querying ATL projects from repositories such as GitHub. Second, we manually identified and removed duplicate scenarios and units from the Zoo. Since this step was not automated, it is possible that some duplicates were missed. Third, we classified modules as endogenous or exogenous based only on the names of the source and target metamodels. Later manual inspection indicated that in some cases, the target metamodel is simply a renamed version of the source, so perhaps these transformations should be categorized as endogenous rather than exongenous. Finally, our study is syntactic, analyzing only the elements of ATL files. Combining it with a semantic or intentional analysis such as [24] might yield deeper insights.

6. Discussion

Based on our experience with ATL and on the findings of this study, we discuss some high level findings that show evidence of the wide applicability of ATL to different problems (Section 6.1). Then, we discuss points that we believe can enhance the experience of ATL users based on our manual inspection of ATL's artifacts, e.g., ATL's grammar and the documentation of the scenarios (Sections 6.2 and 6.3).

6.1. Applicability of ATL

In addition to being used to develop model-to-model (M2M) transformations, ATL has been used in several scenarios to implement transformations that involve text as input or output. For example, the scenario *MicrosoftOffice-ExcelExtractor* uses a query to implement a model-to-text (M2T) transformation from Excel XML models to Excel XML textual files that can be manipulated by Microsoft Excel. In fact, we found that the main use of queries in the Zoo is to implement M2T transformations that generate textual files (e.g., code or XML files) from models, as opposed to generating a single value. For text-to-model (T2M) transformations, some scenarios implemented them using programs in other languages, and then manipulated the output model using an ATL transformation. For example, the scenario *PathExp2PetriNet* uses a TCS (Textual Concrete Syntax) program to transform a textual *path expression* into an equivalent path expression model conforming to the *TextualPathExp* metamodel. The output path expression model is then transformed into a Petri net model using ATL.

We also found scenarios that chain several modules and/or queries to achieve a certain goal. For example, the scenario *MySQL2KM3* chains three modules; the first preprocesses XML models by removing empty elements, the second transforms the preprocessed XML models to MySQL models, and the third transforms MySQL models

to KM3 models. Some of the chains we found to be *non-linear*, i.e., the output of one transformation was used as the input to several transformations, or the outputs of several transformations were used as the inputs to one transformation. The scenario *MeasuringModelRepositories* is one such non-linear transformation chain. The scenario contains a transformation that generates a table model containing measurement information of KM3 models, followed by three transformations that convert the table model into models representing different visual formats (i.e., SVG bar charts, SVG pie charts, and tabular HTML forms). After manually inspecting the scenarios and their documentation, we found that 44 scenarios out of the 99 are transformation chains. We only consider a scenario to be a transformation chain if it chains ATL modules or queries. However, if a scenario chains an ATL module with a plugin or a program in another language, we do not consider it a chain in this study. Moreover, 19 scenarios had several implementations of the same transformation. These scenarios were not considered transformation chains, since their constituent transformations are not meant to be executed sequentially.

Bidirectional transformation languages facilitate building transformations that can be executed in two directions; from the input to the output, and vice versa [16]. ATL is not inherently bidirectional, unlike triple graph grammars for example [25]. Nevertheless, ATL has been used to manually implement five bidirectional scenarios in the Zoo, i.e., these scenarios had two transformations (or transformation chains) to transform an input to an output, and vice versa. For example, the scenario PathExp2Petrinet has six modules; three are a forward transformation chain from textual definitions of path expressions to their equivalent XML representation of Petri nets, and three are the reverse transformation chain.

As shown in our previous discussion, ATL is a versatile language and has been used to solve different transformation problems. While originally designed to be a unidirectional M2M transformation language, the scenarios in the Zoo have used ATL to develop bidirectional transformations as well as M2T or T2M transformations. Further, the language has been used to solve complex transformation problems, that span many intents, by chaining transformations together.

6.2. Consistency of Documentation and Grammar

After inspecting ATL's documentation [17] and grammar [21], we identified two inconsistencies between the artifacts. First, according to ATL's documentation [17], source and target patterns are mandatory in matched rules, while the local variables section and the imperative block are optional. Lazy and unique lazy rules are similar, but with the addition of the keywords **lazy rule** or **unique lazy rule** to the rule's declaration. However, according to ATL's grammar [21], the target pattern is optional. Listing 7 shows a snippet of the Eclipse ATL grammar [21], showing the syntax of the three rule types, where they all appear to have an optional target pattern (denoted by '*outPattern?*'). We observed in Section 4 (Fig. 5) that while 99.9% of the matched rules have target patterns, not all of them do, confirming that target patterns are optional in the ATL implementation.

```
matchedRule ::=
    lazyMatchedRule | matchedRule_abstractContents;
lazyMatchedRule ::=
    'unique'? 'lazy' 'abstract'? 'refining'?
    'rule' IDENTIFIER ('extends' IDENTIFIER)?
    '{' inPattern
        ('using' '{' ruleVariableDeclaration* '}')?
        outPattern? actionBlock?
    '}';
matchedRule_abstractContents ::=
    'nodefault'? 'abstract'? 'refining'?
    'rule' IDENTIFIER ('extends' IDENTIFIER)?
    '{' inPattern
        ('using' '{' ruleVariableDeclaration* '}')?
        outPattern? actionBlock?
    '}';
```

Listing 7. A snippet of the Eclipse ATL grammar showing the syntax of three rule types, where the target pattern appears optional (denoted by '?')

Second, the ATL documentation states that current ATL supports the definition of attribute helpers in modules and queries, but not in libraries [17]. However, the ATL grammar [21] allows attribute helpers in all units (including libraries), and in Section 4 (Fig. 1) we observed that 38% of Zoo libraries contain at least one attribute helper.

While keeping the artifacts of a language (e.g., documentation and grammar) consistent over time is not easy, doing so is crucial to ensure that users can take full advantage of the language features. Updating ATL's documentation to address these inconsistencies will enable more users to take full advantage of the capabilities of ATL.

6.3. Features that Require Clarification

We found two aspects that we believe require clarification in the ATL documentation. In Section 4, we discussed that modules can be simultaneously endogenous and exogenous, i.e., ATL allows endogenous, refining modules, with output models conforming to the source metamodel, to additionally have side-effect output models that conform to a different metamodel. We found this feature to be confusing. The ATL documentation [17] says, *"Obviously, refining mode may only be used for endogenous transformations, i.e. when source and target model share the same metamodel."*. The fact that ATL supports simultaneously endogenous-exogenous transformations is a significant advantage for some transformation problems. However, only two modules (in the same scenario) in the Zoo actually use this feature, since it does not appear in the documentation and thus developers are not aware of it. This feature needs to be documented so that developers can take full advantage of it.

Another aspect that seems to require clarification is the difference between declarative and imperative rule types. The ATL documentation [17] says that matched, lazy, and unique lazy rules are declarative, and that regular, entrypoint, and endpoint called rules are imperative. In Section 4 (Fig. 5), imperative blocks were mainly found in regular, entrypoint, and endpoint called rules, consistent with the documentation. However, while unique lazy rules had no imperative blocks, a small percentage of both matched and lazy rules were actually found to contain imperative blocks. Thus, since ATL allows matched/lazy/unique lazy rules to

have imperative blocks, categorizing these rule types as *declarative* in general is confusing to the user.

7. Related Work

Czarnecki and Helsen [26], [27] have proposed design aspects for transformation languages (e.g., rule scoping, directionality), and both Jouault [28] and Huber [29] have evaluated ATL with respect to these aspects. Koch [30] proposed similar design aspects for transformation approaches relevant to UML-based web engineering.

To the best of our knowledge, transformation language features have been studied only a couple of times. Tairas and Cabot [31] describe an Eclipse-based framework for analyzing domain-specific languages for feature occurrence, co-occurrence, and clone detection, which they have used to gather statistics on the use of features in the ATL Zoo. While the quantitative analysis is somewhat similar to our own, our work adds a qualitative analysis of the results, a more refined in-context analysis of features rather than simple co-occurrence, and a deeper analysis of the use of imperative and declarative features to enable verification. While they process only modules, we analyze all ATL units, including libraries and queries, by processing the original ATL source rather than its EMF representation. We also analyze feature use at a higher level, comparing entire scenarios. Finally, we document our data cleaning steps and publish the cleaned data set [19] to allow replication of our study. Kusel et al. [3] also employ the Zoo to determine how frequently different reuse mechanisms are used in ATL. They find that functions are the most frequently used mechanism, followed by rule inheritance, transformation orchestration and higher-order transformations. Our work is also concerned with syntactic features directly supporting reuse (functions, inheritance) with similar results. However, we do not consider 'higher-level' reuse mechanisms (e.g., higher-order transformations, transformation product lines), while [3] ignores features not related to reuse. The work in [4] counts the use of transformation patterns as identified in [32] in ATL, QVT-R, and UML-RSDS transformations found in the literature. In contrast to our work, language features are not considered.

Programming languages on the other hand have been the subject of empirical studies of feature use much more frequently. Several studies have analyzed and compared the design of different programming languages [6], [7], [8], and Uesbeck et al. [5] have surveyed empirical studies of programming languages. Callaú et al. [33] analyzed the use of dynamic features in Smalltalk, concluding that less than 2% of Smalltalk methods use dynamic features, and that the two most popular are already supported in static languages.

Dyer et al. [34] examined the use of 18 Java features in over 31,000 open source projects. They found that some features were heavily used while others are rarely adopted, e.g., in many cases new features could have been used, but were not. Parnin et al. [35] used 40 open source Java projects to investigate how readily new features such as generics are adopted, which factors hamper adoption, and whether the claimed benefits were realized, concluding that such features are only rarely adopted into existing code. Kim et al. [36] used 20 open source C# projects to extend the work in [35] to investigate how readily generics are used in C# in comparison to Java, with similar conclusions.

The use of build systems has also been studied. Martin et al. [37] investigated how GNU Make features are used in over 6,000 hand-written and automatically generated Make files. They found that only the core features of Make are necessary, that hand-written scripts are more likely to use advanced features, and that automatically generated Make files are more likely to depend on undesirable/obsolete features. Zadok [38] studied the complexity of build processes that use conditional compilation and special-purpose macros.

Analysis of programming languages and systems has been used to better understand how developers use language features and thus inform language evolution. This preliminary study is aimed at providing similar insights for model transformation languages, beginning with ATL. We hope that this first step will help kindle a broader interest in empirical analysis of modelling and model transformation languages, to provide better guidance for their evolution.

8. Conclusion and Future Work

We have investigated the use of various ATL features to better understand how is ATL used and to identify aspects that can help evolve the language. To do so, we formulated three research questions that cover several aspects about the use of ATL. Then, we developed an ATL parser using TXL, and we ran the parser on a preprocessed version of the ATL Zoo. Our results revealed several interesting findings with respect to the popularity of different features, the context in which the features are used, and how widespread the use of the imperative subset of ATL is. We also discussed additional observations based on our manual inspection of the Zoo, ATL documentation, and ATL grammar.

We identify six lines of future work. First, adding industrial transformations in our study is crucial to be able to generalize our findings. Second, it would be interesting to determine if ATL could be made more user-friendly through the introduction of high-level features that summarize collections of jointly used lower-level features. Third, identifying popular combinations of imperative features and their declarative equivalents might facilitate the translation and subsequent verification of imperative transformations. Fourth, automatically translating iterative target patterns into (unique) lazy rules could help avoid the use of deprecated constructs. Fifth, the creation of a repository for transformations in different languages would facilitate the analysis and comparison of other transformation languages. Finally, determining if the use of language features is influenced by the purpose or intent of the transformation [24] would add a potentially useful semantic dimension to our analysis.

Acknowledgments

The authors would like to thank AtlanMod's support team for their prompt and useful responses to our questions. This work is supported in part by NSERC grant number CREATE-397879-2011.

References

[1] P. Mohagheghi, W. Gilani, A. Stefanescu, M. A. Fernández, B. Nordmoen, and M. Fritzsche, "Where does Model-driven Engineering Help? Experiences from Three Industrial Cases," *Software and System Modeling (SoSym)*, vol. 12, no. 3, pp. 619–639, 2013.

[2] J. Whittle, J. E. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal, "A taxonomy of tool-related issues affecting the adoption of model-driven engineering," *Software and System Modeling (SoSyM)*, vol. 16, no. 2, pp. 313–331, 2017.

[3] A. Kusel, J. Schoenboeck, M. Wimmer, W. Retschitzegger, W. Schwinger, and G. Kappel, "Reality Check for Model Transformation Reuse: The ATL Transformation Zoo Case Study," in *Workshop on the Analysis of Model Transformations (AMT)*, 2013.

[4] K. Lano, S. Kolahdouz-Rahimi, S. Yassipour-Tehrani, and M. Sharbaf, "A survey of model transformation design pattern usage," in *Intl. Conference on the Theory and Practice of Model Transformation (ICMT)*. Springer, 2017, pp. 108–118.

[5] P. M. Uesbeck, A. Stefik, S. Hanenberg, J. Pedersen, and P. Daleiden, "An Empirical Study on the Impact of C++ Lambdas and Programmer Experience," in *Intl. Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 760–771.

[6] S. Nanz and C. A. Furia, "A Comparative Study of Programming Languages in Rosetta Code," in *Intl. Conference on Software Engineering (ICSE)*. IEEE Press, 2015, pp. 778–788.

[7] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A Large Scale Study of Programming Languages and Code Quality in Github," in *ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 155–165.

[8] A. Stefik and S. Siebert, "An Empirical Investigation Into Programming Language Syntax," *ACM Transactions on Computing Education (TOCE)*, vol. 13, no. 4, p. 19, 2013.

[9] Eclipse, "Atlas Transformation Language (ATL)," available online, http://eclipse.org/atl/.

[10] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A Model Transformation Tool," *Science of Computer Programming*, vol. 72, no. 1, pp. 31–39, 2008.

[11] Eclipse, "ATL Model Transformation Zoo," available online, http://www.eclipse.org/atl/atlTransformations/.

[12] J. R. Cordy, "The TXL Source Transformation Language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, August 2006.

[13] F. Büttner, M. Egea, and J. Cabot, "On Verifying ATL Transformations Using 'Off-the-Shelf' SMT Solvers," in *Intl. Conference on Model Driven Engineering Languages and Systems (MODELS)*. Springer, 2012, pp. 432–448.

[14] F. Büttner, M. Egea, J. Cabot, and M. Gogolla, "Verification of ATL Transformations Using Transformation Models and Model Finders," in *Intl. Conference on Formal Engineering Methods (ICFEM)*. Springer, 2012, pp. 198–213.

[15] F. Büttner, M. Egea, E. Guerra, and J. De Lara, "Checking Model Transformation Refinement," in *Intl. Conference on Theory and Practice of Model Transformations*. Springer, 2013, pp. 158–173.

[16] T. Mens and P. Van Gorp, "A Taxonomy of Model Transformation," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 152, pp. 125–142, 2006.

[17] Eclipse, "ATL User Guide," available online, http://wiki.eclipse.org/ATL/User_Guide_-_Introduction.

[18] ——, "EMFTVM Documentation," available online, https://wiki.eclipse.org/ATL/EMFTVM.

[19] G. M. K. Selim, J. R. Cordy, and J. Dingel, "Preprocessed ATL Zoo," available online, https://github.com/gehanselim/preprocessed-ATL-zoo.

[20] J. R. Cordy, "ATL Grammar for TXL," 2016, available online, http://www.txl.ca/nresources.html.

[21] M. Tisi and F. Jouault, "ATL Grammar," available online, https://wiki.eclipse.org/M2M/ATL/Syntax.

[22] K. Anastasakis, B. Bordbar, and J. M. Küster, "Analysis of Model Transformations via Alloy," in *Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa)*, 2007, pp. 47–56.

[23] J. Cabot, R. Clarisó, E. Guerra, and J. De Lara, "Verification and Validation of Declarative Model-to-Model Transformations Through Invariants," *Journal of Systems and Software*, vol. 83, no. 2, pp. 283–302, 2010.

[24] L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. M. Selim, E. Syriani, and M. Wimmer, "Model Transformation Intents and Their Properties," *Software & Systems Modeling (SoSyM)*, vol. 15, no. 3, pp. 647–684, 2016.

[25] A. Schürr, "Specification of Graph Translators with Triple Graph Grammars," in *Graph-Theoretic Concepts in Computer Science*, ser. LNCS. Springer, 1995, vol. 903, pp. 151–163.

[26] K. Czarnecki and S. Helsen, "Classification of Model Transformation Approaches," in *OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, vol. 45, no. 3. USA, 2003, pp. 1–17.

[27] ——, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006.

[28] F. Jouault and I. Kurtev, "Transforming Models with ATL," in *Intl. Conference on Model Driven Engineering Languages and Systems (MODELS'05)*. Springer, 2005, pp. 128–138.

[29] P. Huber, "The Model Transformation Language Jungle: An Evaluation and Extension of Existing Approaches," Master's thesis, Vienna University of Technology, 2008.

[30] N. Koch, "Classification of Model Transformation Techniques Used in UML-Based Web Engineering," *IET Software*, vol. 1, no. 3, pp. 98–111, 2007.

[31] R. Tairas and J. Cabot, "Corpus-Based Analysis of Domain-Specific Languages," *Software & Systems Modeling (SoSyM)*, vol. 14, no. 2, pp. 889–904, 2015.

[32] K. Lano and S. Kolahdouz-Rahimi, "Model-transformation design patterns," *Trans. Software Eng.*, vol. 40, no. 12, pp. 1224–1259, Dec 2014.

[33] O. Callaú, R. Robbes, É. Tanter, and D. Röthlisberger, "How Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk," in *Intl. Working Conference on Mining Software Repositories (MSR)*, 2011.

[34] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features," in *Intl. Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 779–790.

[35] C. Parnin, C. Bird, and E. Murphy-Hill, "Adoption and Use of Java Generics," *Empirical Software Engineering*, vol. 18, no. 6, pp. 1047–1089, 2013.

[36] D. Kim, E. Murphy-Hill, C. Parnin, C. Bird, and R. Garcia, "The Reaction of Open-Source Projects to New Language Features: An Empirical Study of C# Generics," *Journal of Object Technology (JOT)*, vol. 12, no. 4, 2013.

[37] D. H. Martin, J. R. Cordy, B. Adams, and G. Antoniol, "Make It Simple: An Empirical Analysis of GNU Make Feature Use in Open Source Projects," in *Intl. Conference on Program Comprehension (ICPC)*. IEEE Press, 2015, pp. 207–217.

[38] E. Zadok, "Overhauling Amd for the'00s: A Case Study of GNU Autotools." in *USENIX Annual Technical Conference, FREENIX Track*, 2002, pp. 287–297.